E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Desarrollo de una interfaz gráfica de usuario para el control de analizadores de espectros ópticos mediante Matlab

Grado en Ingeniería
en Tecnologías Industriales

Trabajo Fin de Grado

Nombre y apellidos del autor: Jon Gomez Labat
Nombre y apellidos del director/es: Ignacio Del Villar Fernández
Pamplona, fecha de defensa: 28 junio 2016

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Agradecimientos:

Al director del proyecto, Ignacio Del Villar, por su paciencia, comprensión y dedicación así como por ayudarme a resolver todas las dudas que me han ido surgiendo a lo largo de la elaboración de este trabajo.

RESUMEN

Este trabajo se ha realizado ante la necesidad que existía en el laboratorio de sensores de la UPNA (UPNA Sensors) del desarrollo de una aplicación gráfica en el entorno de desarrollo integrado (IDE) proporcionado por la herramienta de software matemático Matlab para el control y obtención de datos mediante los analizadores "EXFO" y "AGILENT 86142A". Así, se ha facilitado tanto la obtención de datos en la realización de mediciones como el control de las principales funciones del analizador de forma remota, buscando simplificar el proceso que hay que llevar a cabo en la medición de la potencia óptica por longitud de onda en el laboratorio. Para ello, se ha buscado la mayor universalidad posible en el desarrollo del software, de tal forma que, mediante una adaptación sencilla del código, se pueda utilizar la interfaz independientemente del modelo o fabricante del analizador, siempre y cuando la comunicación con el instrumento de medida se realice mediante el estándar IEE 488.2.

ABSTRACT

In this work, a graphical user interface for control and data acquisition with optical spectrum analyzers was developed using the multi-paradigm numerical computing environment provided by Matlab. The main objective is to ease the acquisition of the spectral power vs wavelength on an optical signal by simplifying the process of initialization, configuration and measure with an OSA. Although this GUI has been configured for both EXFO and AGILENT 86142A analyzers, the software design has a universal approach so it can be extended to any instrument which uses the IEE 488.2 communication standard regardless of manufacturer. This adaptation and extension of the developed software can be made by a simple code modification, following the structure stated in this work.

Índice de contenidos

1. Introducción.....1

2. Objetivos. 3

3. Estado del arte. 5

3.1 Nivel hardware. 5

3.2 Nivel software. 9

4. Metodología. 11

4.1 Nivel hardware. 11

4.2 Nivel software. 13

4.3 Desarrollo del código. 14

5. Desarrollo y descripción de la interfaz de usuario. 15

5.1 Comprensión del programa heredado y establecimiento de las necesidades mínimas que debe satisfacer el código que se va a desarrollar. 15

5.2 Necesidades mínimas que debe satisfacer el código. 16

5.3 Descripción detallada del software desarrollado. 17

5.3.1 Organización y estructura general. 17

5.3.1.1 Diagrama de flujo principal. 18

5.3.1.2 Desglose de las distintas funcionalidades de la interfaz. 19

5.3.1.3 Interacción con el usuario. 22

5.3.1.4 Estructura de los objetos gráficos y eventos de ejecución desarrollados. 23

5.3.1.5 Estructura de variables principal y definición dinámica de la misma. 24

5.3.2 Funciones comunes 26

5.3.2.1 Inicialización del software. 26

5.3.2.2 Ventana principal de la interfaz. 29

5.3.2.3 Función de dialogo. 39

5.3.2.4 Función de cargado de referencias. 40

5.3.2.5 Función de gestión de referencias. 42
5.3.2.6 Generación de tramos para el zoom en las medidas. 46
5.3.2.7 Funciones de control de medida. 48
5.3.2.8 Función de guardado de datos. 50
5.3.2.9 Exportación de datos. 52
5.3.3 Funciones propias de cada analizador. 57
5.3.3.1 Función de inicialización del OSA. 57
5.3.3.2 Función de configuración del OSA. 60
5.3.3.3 Función de medida. 60
5.3.3.4 Ventana de parámetros. 62
5.3.3.5 Función principal de gestión del analizador. 64
5.3.3.6 Procesado de los datos. 67
5.3.4 Gestión de errores y excepciones durante la ejecución del software. 68
6. Resultados y demostración del funcionamiento de la interfaz. 69
6.1 Configuración utilizada. 69
6.2 Procedimiento de medida. 69
6.3 Resultados. 72
7. Problemáticas encontradas en el uso de Matlab como entorno de desarrollo 73
8. Conclusiones 76
9. Líneas futuras. 77
9.1 Ampliación de la interfaz a un nuevo analizador. 77
9.2 Desarrollo de una interfaz en Python 3.4 mediante el uso de la biblioteca Qt. 78
9.2.1 Introducción 78
9.2.2 Preparación del entorno de trabajo. Instalación del software necesario para comenzar el desarrollo. 79
9.2.3 Estructura del código y jerarquización de clases. 79
9.2.4 Establecimiento de la comunicación y utilización del módulo pyVISA mediante el

módulo “communicate.py” desarrollado.....80

9.2.5 Configuración del OSA y manejo de las instrucciones SCPI. 82

9.2.6 Desarrollo de las funciones necesarias para la creación de los objetos y manejo de los mismos. 85

9.2.7 Desarrollo de la interfaz y manejo de las funciones definidas en el módulo “main.py”. .88

9.2.8 Muestra del funcionamiento de la interfaz desarrollada en Python 3.4 89

9.3 Guardado de datos en un servidor 90

10. Bibliografía y referencias. 91

11. Anexos 93

11.1 Instrucciones SCPI, para la inicialización, configuración y medida necesarias para los modelos de OSA configurados. 93

11.2 Desglose de la estructura de variables “data”. 94

11.3 Código correspondiente a la función principal para la interfaz comenzada a desarrollar en Python 3.4. 97

1. Introducción.

La finalidad fundamental del trabajo desarrollado es la de facilitar la obtención de datos mediante un analizador de espectros ópticos (Optical Spectrum Analyzer - OSA). La obtención de dichas medidas y la comunicación con los OSA se realiza a partir de la IDE proporcionada por Matlab, haciendo uso del módulo GPIB. El punto de partida de la interfaz es la obtención de los datos a partir de un código cuya única interacción con el usuario ocurre ante la modificación del propio código, es decir, que es necesario rehacer o retocar los parámetros del programa cada vez que se desea realizar una medida distinta. Este programa de partida, fue desarrollado para suplir la falta de una interfaz de control amigable integrada en los OSA. Por ello, éste código se amplia y mejora para alcanzar la característica principal de la interfaz de usuario desarrollada en el trabajo: controlar el OSA de una forma cómoda y sin necesidad de modificar el código utilizado para el control y comunicación del aparato cada vez que se desea realizar una nueva medida.

A lo largo de estas líneas, se cita y se hace uso de terminología técnica relacionada con la fibra óptica y referida al uso al que se ha orientado la interfaz. Dichos conceptos y elementos se utilizan únicamente en forma de herramienta y de ningún modo son el foco principal de este trabajo ni se pretende ir más allá en este aspecto de lo que se desarrolla a lo largo de estas páginas.

El presente documento se ha estructurado como una memoria monográfica y no pretende únicamente ser un texto descriptivo del trabajo técnico realizado durante el Trabajo de Fin de Grado (TFG), sino además poner en contexto las problemáticas encontradas durante la realización del mismo y proponer soluciones que puedan ser desarrolladas en el futuro.

Se partirá de unos objetivos propuestos con anterioridad al desarrollo de la interfaz y se analizará la situación actual existente, tanto a nivel *hardware* como *software*. Acto seguido, se establecerán unas necesidades que deberá satisfacer la interfaz implementando las características y funcionalidades que se consideren oportunas tanto para cumplir las especificaciones técnicas propuestas como para facilitar su manejo.

Teniendo en cuenta todos estos aspectos, se procederá a realizar una redacción descriptiva del desarrollo de la parte técnica del trabajo, que constará de tres secciones. La primera de ellas, es la sección *Metodología empleada* que trata de responder a la pregunta ¿qué ha sido necesario

utilizar y cómo se ha utilizado para el desarrollo de la interfaz? La sección *Desarrollo y descripción de la interfaz de usuario* se centra en la implementación práctica en Matlab de todo lo anterior, explicando la funcionalidad técnica de la interfaz y describiendo las dificultades encontradas y cómo se han solventado. Finalmente, la sección *Resultados y demostración de funcionamiento* describe un ejemplo práctico de uso de la interfaz.

Por último, se exponen las conclusiones obtenidas durante la realización del trabajo, se describen las problemáticas encontradas durante el mismo y se enlaza con la proposición de una serie de líneas futuras que intentan no solamente resolver dichos inconvenientes sino ampliar y mejorar las funcionalidades de la interfaz desarrollada.

Esta interfaz permite, por ejemplo, realizar una configuración del rango de longitudes de onda en los que se desea obtener la medida de la potencia del espectro, introduciendo los parámetros de *span* y *center* deseados. Estas órdenes que se envían al instrumento son completamente transparentes para el usuario. Además, se pueden configurar otros parámetros útiles que se detallarán en la sección *Desarrollo y descripción de la interfaz de usuario*.

Finalmente, cabe destacar la posibilidad de ampliación tanto a nuevos instrumentos como a nuevos parámetros de configuración dentro de cada OSA. Es decir, tal y como se ha organizado el código, sería posible, sin modificar la parte constructiva de la interfaz, que se encarga de comunicarse con el usuario tanto para la introducción de ordenes como para mostrar los resultados de las medidas realizadas, añadir menús de configuración nuevos que permitan configurar otros parámetros del instrumento o incluso controlar un nuevo instrumento.

2. Objetivos.

Inicialmente, el objetivo del trabajo era conseguir una comunicación satisfactoria para poder controlar el analizador EXFO, que debido a su pobre interfaz incorporada y a su antigüedad presenta un manejo de gran dificultad, incómodo y muy poco agradable al usuario. Por ello, dificulta su utilización práctica para realizar medidas. También se propuso implementar dicha comunicación y control en una interfaz gráfica que supliese las graves carencias de este aparato en cuanto a su manejo. Más adelante, se decidió ampliar este objetivo para lograr una inclusión, en el software de control, del OSA Agilent que se utiliza más a menudo en el laboratorio, siendo esta inclusión una segunda parte del desarrollo del trabajo.

Para poder lograr el objetivo principal, fue necesario comprender cómo se realiza la comunicación a nivel hardware entre el aparato de medida y el PC. Esta comunicación se realiza mediante el estándar de bus de datos digital de corto rango GPIB desarrollado por Hewlett-Packard en los años 70 para la conexión de dispositivos de medida con dispositivos de control [1]. El bus fue estandarizado por el *Institute of Electrical and Electronics Engineers* y es este protocolo el que se ha utilizado.

Para comunicar el dispositivo de control (PC) con el OSA fue necesario instalar y configurar adecuadamente los drivers de la tarjeta GPIB ya instalada en el dispositivo de control previamente a la realización de este trabajo.

La comunicación con el OSA, y la sintaxis necesaria se encuentra detallada en el SCPI (*Standard Commands for Programmable Instruments*) [2]. Las instrucciones SCPI son las que realmente contienen las órdenes que serán aceptadas y comprendidas por el instrumento que se vaya a controlar y son las que manejará en última instancia el código de Matlab. Por ello, antes de desarrollar un software que maneje estas instrucciones y se comunique satisfactoriamente con el instrumento a través de ellas, es necesario seleccionar y comprobar cuáles de estas instrucciones son compatibles o *entendidas* por el instrumento a controlar, ya que SCPI es un estándar que engloba multitud de órdenes debido a su carácter universal. Este estándar se desarrolló con el objetivo de reducir el tiempo de desarrollo de un programa de *Automatic Test Equipment* (ATE) [3]. En el manual de cada instrumento es posible encontrar qué instrucciones se deben utilizar (ver anexos, sección *10.1. Instrucciones SCPI para la inicialización, configuración y medida necesacias*

para los modelos de OSA configurados).

Es imprescindible e importante, por todo ello, realizar un trabajo exhaustivo de adquisición de información a la hora de desarrollar cualquier tipo de código destinado al control y obtención de datos de dichos dispositivos ya que las órdenes o comandos tanto de configuración como de control varían de un instrumento a otro.

Una vez recopilada la información necesaria, presente en los manuales de dichos instrumentos, se está en disposición de comenzar a adecuar dichas órdenes a las necesidades que se deseen satisfacer. En este caso será facilitar un manejo del instrumento mucho más intuitivo para el usuario.

3. Estado del arte.

3.1 Nivel hardware.

En primer lugar, se realizará un análisis de los instrumentos que se utilizan para medir espectros ópticos y que se pueden encontrar en el mercado actualmente. En concreto se va a diferenciar entre espectrómetros y analizadores de espectros ópticos (OSAs). Para ello se han seleccionado las principales firmas en estos ámbitos: OceanOptics, Spectral Products y Thorlabs.

Un analizador de espectro óptico (OSA) es un instrumento que mide la magnitud de la potencia de una señal óptica en función de la frecuencia, teniendo un rango de frecuencia de trabajo. Debido al orden de magnitud de las frecuencias con las que se trabaja en el uso de analizadores, el parámetro que se utiliza no es la frecuencia sino la longitud de onda, que resulta mucho más manejable. La señal óptica se introduce directamente en el instrumento, que utiliza técnicas reflectoras y/o refractarias para separar las distintas longitudes de onda y después mide la intensidad de la luz mediante un detector óptico-electrónico. Existen varios métodos para separar las distintas longitudes de onda y no se profundizará en ellos, sin embargo, una de las características más importantes y que más peso tiene en el coste económico de un OSA es la resolución que presenta, es decir, la mínima variación de longitud de onda que es capaz de distinguir.

Se encuentran distintos tipos de analizadores, que se diferencian, excluyendo la resolución que tengan, por los distintos rangos de longitudes de onda que pueden detectar. Una clasificación general tendría dos apartados, uno en el que se agrupan instrumentos optimizados para medir en el orden de nanómetros y otro que engloba aquellos OSA que miden en el orden de micrómetros. Desde el mismo punto de vista de rangos de longitudes de onda, la distinción lógica que se realiza es entre instrumentos que miden en la zona visible (300-1000nm) e instrumentos que operan en la zona infrarroja (1000-1800nm).

En la búsqueda realizada, se ha llegado a la conclusión de que actualmente existen más compañías dedicadas al desarrollo de espectrómetros que de analizadores ópticos. Un espectrómetro es un instrumento de medida que además de medir la magnitud de potencia espectral de una señal óptica, está diseñado para medir otras propiedades, como la polarización electromagnética, siempre en un rango determinado. No obstante, los analizadores de espectros

ópticos ofrecen resoluciones mayores.

El hecho de que estos instrumentos únicamente estén diseñados para medir en unos determinados rangos de longitud de onda se debe a las distintas técnicas utilizadas para medir las distintas secciones del espectro óptico. La detección se realiza mediante un foto-detector, que constituye el elemento de entrada de cualquier receptor óptico y convierte la luz en corriente eléctrica. En el caso de sistemas de fibra óptica los materiales base que se utilizan para su fabricación son semiconductores. Existen dos tipos de foto-detectores fundamentales, los térmicos y los fotónicos. Los térmicos absorben la energía de los fotones incidentes en forma de calor, cambiando las propiedades eléctricas (resistencia) del material en función de la temperatura. Los detectores fotónicos utilizan la energía del fotón para modificar las propiedades de conducción eléctrica. Estos detectores son los que se utilizan más a menudo en sistemas de comunicaciones y se caracterizan mediante los siguientes parámetros fundamentales: la eficiencia cuántica, la responsividad, el tiempo de respuesta y las características de ruido [4].

Considerando la estructura de un semiconductor como se ve en la *Figura 1*, si la energía de los fotones incidentes excede la de la banda prohibida y bajo la influencia de un campo eléctrico, aparece una corriente (fotocorriente), I_F, proporcional a la potencia óptica incidente P_{IN}. La constante de proporcionalidad que relaciona la corriente con la potencia óptica incidente es la responsividad del foto-detector, R [4].

$$I_F = R \cdot P_{IN}$$

La eficiencia cuántica es la relación entre el flujo de electrones generado con la tasa de incidencia de fotones. El tiempo de respuesta se define como el intervalo de tiempo que, ante una variación brusca de la potencia óptica, la corriente I_F tarda en aumentar hasta el 90% de su valor final. Por último, un foto-detector presenta diversas fuentes de ruido que se observan como una corriente aleatoria en torno a su valor medio. Las fuentes de ruido habituales, son la llegada al detector de fotones no deseados, la corriente de oscuridad, el ruido de ganancia y todo aquel ruido que produzcan los componentes de los circuitos del propio receptor [4].

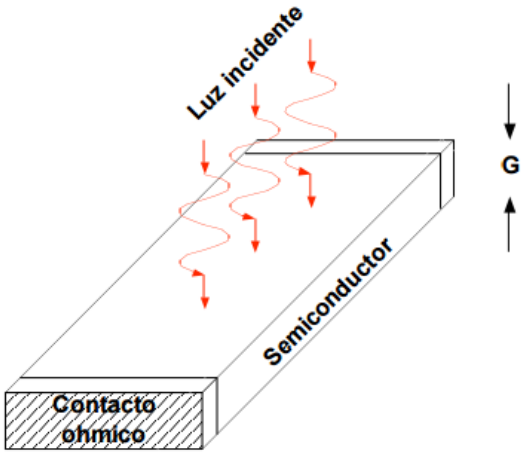

Figura 1. Estructura semiconductora utilizada como foto-detector [4].

A modo de ejemplo, se ha seleccionado para esta explicación un espectrómetro de la firma Thorlabs (*Figura 2*), que permite realizar mediciones en el rango del espectro visible, y un espectrómetro de la firma OceanOptics (*Figura 3*), que presenta un rango de medida situado en el espectro infrarrojo. Se citan así mismo, las tres características más importantes de estos instrumentos, el rango en el que mide, su resolución y el tiempo de integración que presenta. La resolución del instrumento se mide en FWHM del inglés, *Full Width at Half Maximum* o anchura a media altura, una medida de la extensión de una función utilizada en la medida de este parámetro en espectrómetros.

También se cita el modelo OSA202 de la firma Thorlabs (*Figura 4*), que mide en el rango entre 600nm y 1700nm. La resolución se especifica en GHz y cm^{-1} y, en la sección "design" de la documentación proporcionada por el fabricante, se calcula mediante la siguiente expresión:

$$\Delta\lambda = \Delta k \times 100 \times \lambda^2$$

Donde Δλ es la resolución en pm, Δ*k* es la resolución en cm^{-1} (con un máximo en 0.25cm^{-1} para este instrumento) y λ es la longitud de onda en µm [5].

Espectrómetro en el rango de luz visible, Thorlabs CCS175 Spectrometer.

Rango de medida: 500-1000nm

Resolución: 0.6nm FWHM

Tiempo de integración: 10 µs – 60 s

Figura 2. Thorlabs CCS175 Spectometer [6].

Espectrómetro en el rango infrarojo, OceanOptics Flame-NIR Spectometer.

Rango de medida: 950-1650nm

Resolución: 10nm FWHM

Tiempo de integración: 1 ms – 60 s

Figura 3. OceanOptics Flame-NIR Spectometer [7].

OSA Thorlabs 202.

Rango de medida: 600-1700nm

Resolución: 7.5GHz (0.25cm^{-1})

Figura 4. Thorlabs OSA202 [8].

Dentro del apartado del hardware, es necesario citar además el tipo de comunicación que se realiza. El bus de comunicación que más se ha utilizado tradicionalmente ha sido el bus de datos digital de corto rango GPIB junto con su estándar IEEE 488.1 que define el hardware (y el estándar IEEE 488.2 que define la sintaxis). Sin embargo, actualmente la comunicación de los instrumentos de medida con el dispositivo controlador se realiza prácticamente en la totalidad de casos vía USB. Cuando es necesario controlar un instrumento que no disponga de una interfaz USB, como es el caso de los instrumentos utilizados en el laboratorio, se hace uso, habitualmente, de un adaptador GPIB-USB, con el objetivo de evitar la instalación de una tarjeta GPIB compatible con el estándar IEEE 488 en las ranuras PCI (*Peripheral Component Interconnect*) presentes en los equipos de control.

3.2 Nivel software.

Si se analizan los programas comerciales y entornos de desarrollo para el control, obtención, análisis y visualización de datos obtenidos mediante instrumentos de medida, se encuentran dos grandes alternativas: LabView, creado por National Instruments, y Matlab, diseñado por MathWorks.

Estas dos herramientas copan los ámbitos tanto académico como científico-investigador. Para la realización de este trabajo se ha utilizado el entorno proporcionado por Matlab que cuenta con un lenguaje de programación propio, y potentes herramientas para la adquisición de datos. En concreto se ha utilizado el módulo GPIB dentro del ICT (*Instrument Control Toolbox*) que crea el objeto de comunicación [9]. El entorno proporcionado por LabView, por el contrario, tiene como característica principal un lenguaje de programación gráfico, que permite al desarrollador operar a un nivel más alto. Tiene incluido, por ejemplo, un bloque VISA mediante el cual se podría configurar fácilmente una conexión a través de un bus GPIB [10]. Esta característica y la posibilidad de realizar una programación a tan alto nivel, aunque puede resultar altamente atractiva ya que facilita el diseño de controladores y sistemas muy complejos reduciendo el tiempo de desarrollo, en este caso resulta excesiva. Se opta por realizar un control a un nivel inferior con el objetivo de alcanzar una mayor estabilidad y precisión en el mismo, así como aprovechar la gestión de procesado numérico que ofrece Matlab. La potencia del motor de Matlab se hace necesaria para lograr una gran fluidez y una alta eficiencia en el manejo de los vectores de datos que se deben procesar.

Por otro lado, como alternativa a estas dos importantes herramientas utilizadas ampliamente en la adquisición y visualización de datos obtenidos por instrumentos de medida, se encuentran otros lenguajes de programación también empleados de forma habitual en la computación científica y en el establecimiento y mantenimiento de comunicaciones, como pueden ser C++ o Python. En concreto, el lenguaje de programación interpretado Python, cumple con todas las características necesarias para poder ser utilizado de una forma sencilla en la realización de todas las tareas y procesos necesarios en el desarrollo de este trabajo (ver apartado *Líneas Futuras*).

Todo el trabajo desarrollado, se ha basado en el programa escrito en lenguaje M (lenguaje propio de Matlab) que se venía utilizando hasta ahora en el laboratorio y que como ya se ha especificado anteriormente carecía de interfaz gráfica de usuario (en adelante *programa heredado*).

4. Metodología.

La metodología empleada en este trabajo puede dividirse así mismo en los dos grupos que ya se han desglosado en el apartado *Estado del arte*.

4.1 Nivel hardware.

A nivel hardware y como herramientas principales se han utilizado los analizadores de espectros ópticos EXFO y AGILENT 86142A para los que se ha desarrollado la interfaz gráfica de usuario. Así mismo, ha sido necesario utilizar la fuente de luz Agilent presente en el laboratorio de sensores y utilizada habitualmente para realizar las medidas.

El analizador EXFO presenta un rango de medida máximo de 1250nm a 1600nm mientras que el OSA Agilent 86142A tiene un rango de medida de 600nm a 1700nm y alcanza una resolución máxima FWHM de 0.06nm. La resolución en el EXFO no se modifica mediante ningún parámetro de configuración, sin embargo, en la configuración del OSA para la realización de las pruebas se ha seleccionado una resolución de 0.5nm FWHM [11].

La conexión entre la fuente de luz y los OSA, para el desarrollo de las pruebas, se ha realizado mediante una fibra óptica LPFG acrónimo de *Long-Period Fiber-Grating.* El *grating* en una fibra, consiste en la perturbación periódica de las propiedades de la fibra, generalmente en el índice refractivo del núcleo. Tal y como se aprecia en la *Figura 5*, el espectro de la luz de entrada es plano, mientras que a la salida de la LPFG se observan las bandas de atenuación características de este tipo de fibra. Cada banda de atenuación presenta un mínimo, definido como la longitud de onda de resonancia.

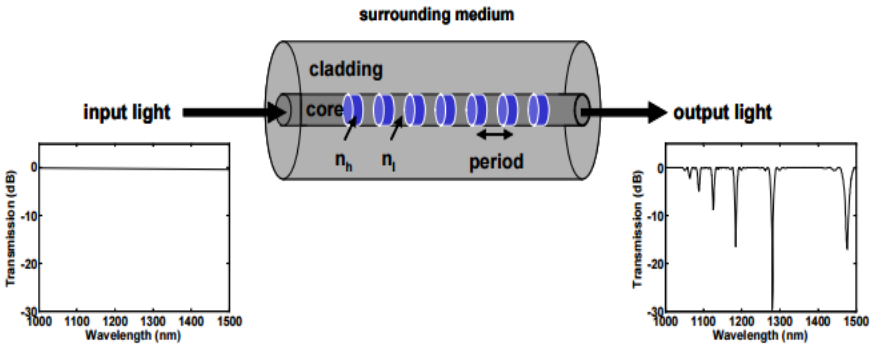

Figura 5. Esquema de una fibra LPFG [12].

Para realizar una aproximación de las longitudes de onda de resonancia, se utiliza una condición de coincidencia de fases modificada tal y como se expresa en la ecuación:

$$\beta_{01}(\lambda)+s_0\zeta_{01,01}(\lambda)-\left(\beta_{0j}(\lambda)+s_0\zeta_{0j,0j}(\lambda)\right)=\frac{2\pi N}{\Lambda}$$

Donde β_{01} y β_{0j} son las constantes de propagación del modo del núcleo y del modo número *j* del cladding respectivamente, $\zeta_{01,01}$ y $\zeta_{0j,0j}$ son los coeficientes de auto-acople del modo del núcleo y del modo *j* del *cladding*, S_0 es el coeficiente de la primera componente de Fourier del *grating,* Λ es el período del *grating* y N es el orden de difracción [12].

La profundidad de las bandas de atenuación depende de la longitud del *grating* y del coeficiente de acoplamiento cruzado entre el modo del núcleo y el del cladding. En la *Figura 6*, se puede observar la evolución del espectro a la salida de una LPFG con el incremento de la longitud del *grating* [12].

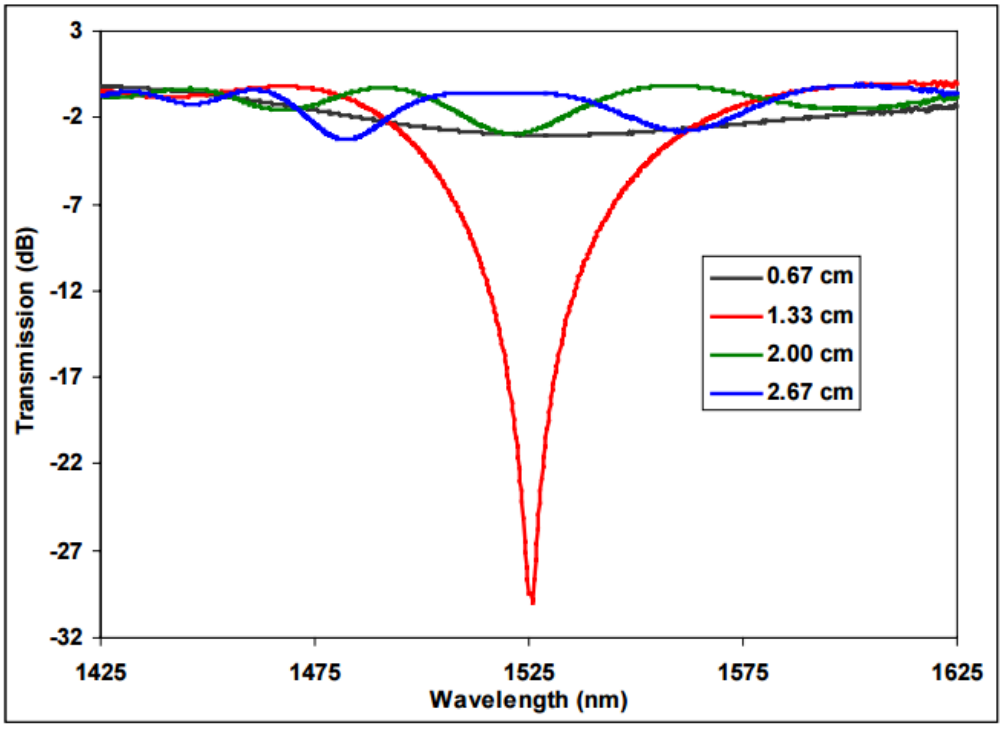

Figura 6. Evolución del espectro a la salida de una LPFG en función de la longitud del "grating" [12].

Es necesario aclarar que la utilización de esta fibra óptica ha sido únicamente como herramienta y con el objetivo de poder realizar medidas que orientasen el desarrollo de la interfaz desarrollada y en ningún caso se ha profundizado en los conceptos técnicos relacionados con este campo y que se mencionan en el párrafo anterior.

Para poder establecer una comunicación efectiva entre el dispositivo de medida y el dispositivo de control mediante el bus GPIB, ha sido necesario así mismo configurar la tarjeta PCI-GPIB integrada en el PC utilizado para el control.

4.2 Nivel software.

Utilizando como enlace con la sección de hardware la tarjeta necesaria para establecer la comunicación a través del bus GPIB, surge la necesidad de contar con un controlador (o *device driver*) que permita al dispositivo de control reconocer y operar con dicha tarjeta. Para la instalación de los drivers de la tarjeta desarrollados por National Instruments, se optó por instalar el paquete NI-MAX [13], que además de contener los controladores necesarios, consta de una interfaz que permite la rápida identificación de las direcciones GPIB asignadas a los distintos instrumentos que estén conectados al dispositivo de control, facilitando el posterior establecimiento de una conexión desde el entorno de desarrollo.

Dentro del IDE de MatLab se ha hecho uso del *Intrument Control Toolbox*, utilizando el módulo GPIB [9].

4.3 Desarrollo del código.

Para la parte principal de este trabajo, la metodología utilizada ha sido un proceso iterativo que puede resumirse a grandes rasgos en el esquema de la *Figura 7*, en el que además se presentan los cuatro grandes bloques en los que se ha centrado el desarrollo.

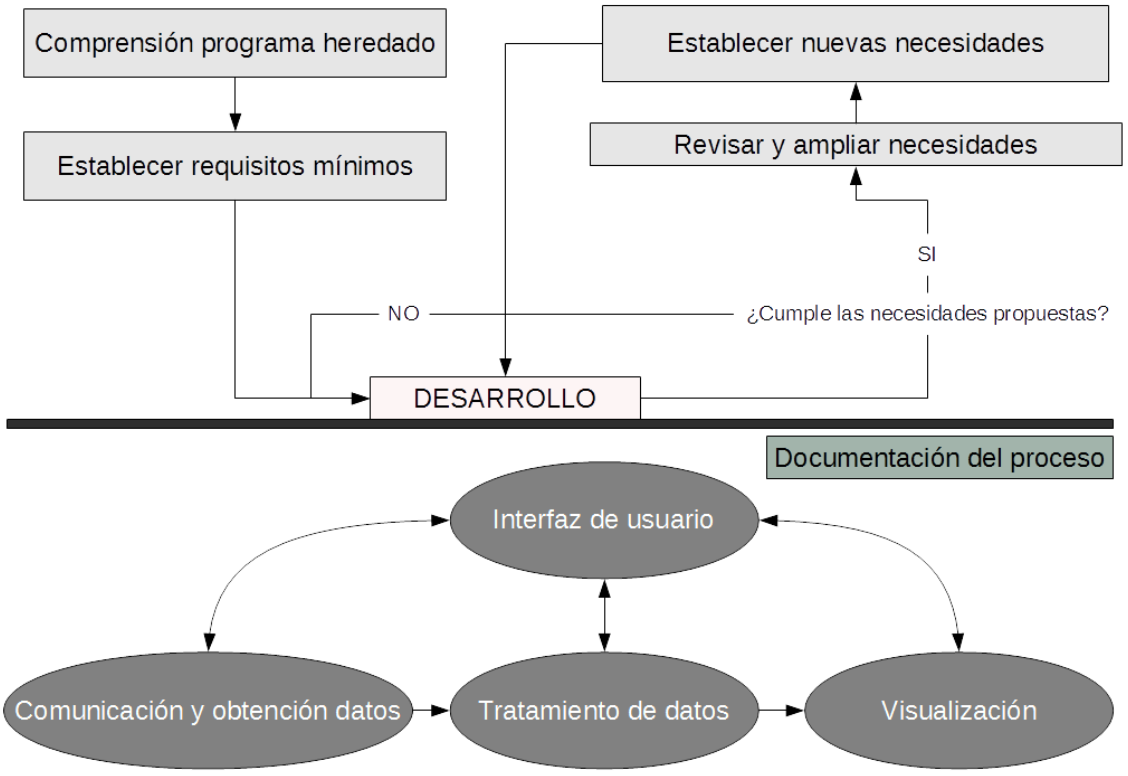

Figura 7. Esquema del proceso iterativo empleado como metodología en el desarrollo del software.

5. Desarrollo y descripción de la interfaz de usuario.

5.1 Comprensión del programa heredado y establecimiento de las necesidades mínimas que debe satisfacer el código que se va a desarrollar.

Siguiendo la metodología descrita en el apartado anterior, y tal y como puede observarse en el esquema de la *Figura 7*, el desarrollo de la interfaz se comienza por una comprensión del programa heredado, así como por el establecimiento de unas necesidades mínimas a partir de este código. Además, como ya se ha especificado anteriormente, el código comienza a desarrollarse en torno al control del analizador EXFO.

Del análisis del programa heredado se identifican tres secciones importantes en el código. La primera de ellas se encarga del establecimiento de una comunicación satisfactoria, de configurar el instrumento y de realizar una medición, es decir, que realiza todas las operaciones en las que está presente el dispositivo de adquisición de datos. La segunda sección realiza un proceso de tratamiento de datos, es decir, se encarga de adecuar la información adquirida y prepararla para mostrarla en la pantalla. La tercera y última sección muestra los datos de una forma ordenada y agradable en forma de gráfica al usuario.

En el diseño de la estructura del código se han mantenido estas tres secciones principales, las cuales se han desarrollado ampliamente para cubrir todas las funcionalidades y requerimientos que se han considerado indispensables en la interfaz. Esta interfaz, por analogía con el análisis realizado anteriormente, no es sino una cuarta sección del código, que sin embargo engloba a las tres secciones ya definidas y se encarga de gestionarlas y adecuarlas a la manipulación de las mismas por parte del usuario.

5.2 Necesidades mínimas que debe satisfacer el código.

Dentro de la sección de comunicación, el programa deberá ser capaz de establecer una comunicación correcta con el instrumento, es decir, ser capaz de enviar las instrucciones necesarias para: iniciar el instrumento, configurarlo a partir de unos parámetros, solicitar los datos de medida y leer la información recibida.

La forma de operar con el analizador de espectros ópticos requiere que se realice en primer lugar un barrido de referencia contra el que se compararán las medidas que se deseen realizar. Así, en la sección de tratamiento de datos, se establecen como necesidades mínimas la comparación de la señal obtenida con la señal de referencia, el suavizado de estos datos con la finalidad de reducir el ruido y facilitar la lectura de la misma, la posibilidad de visualizar estos datos obtenidos y la transformación de los datos ya procesados en dos vectores. Estos vectores contendrán la medida del espectro que se desea visualizar (potencia espectral) y las longitudes de onda en las que se ha medido. El vector de longitudes de onda no es sino la discretización del espacio lineal formado por los valores comprendidos entre los dos extremos que delimitan el rango de longitudes de onda en el que se ha tomado la medida. Este rango tiene una variación entre los valores definida por el parámetro *delta* cuyo valor viene dado por el número de puntos obtenidos en la medida. Esta variable deberá ser establecida en la configuración del analizador de espectros y presenta un límite en la resolución máxima que tenga el instrumento.

La sección de visualización de datos, se encarga de presentar en forma de gráfico la información proporcionada por la sección de procesado. Se definen dos necesidades de representación. La primera de ellas es un eje de coordenadas en el que se dibujará la potencia espectral frente a las longitudes de onda y la segunda de ellas deberá procesar las medidas realizadas y elaborar un mapa de colores en el que se presentan las longitudes de onda frente al número de vectores de datos, definiendo una escala de colores en función de la potencia espectral.

Todas estas especificaciones deberán ser cumplidas dentro de una interfaz gráfica compacta que se comunique con el usuario, permita configurar el analizador de espectros e incluya los espacios necesarios para contener las ventanas de visualización de datos.

Una vez satisfechas las necesidades mínimas anteriormente establecidas, se inicia un proceso iterativo de desarrollo en el que se plantean nuevas necesidades que se van incorporando al software, tal y como se observa en el esquema general presentado en la sección *4. Metodología*.

5.3 Descripción detallada del software desarrollado.

5.3.1 Organización y estructura general.

La estructura general del software se ha desarrollado a partir de un directorio raíz en el que se localizan todas aquellas funciones y archivos de código de carácter común, es decir, que son utilizados por la interfaz independientemente de qué analizador de espectros se esté utilizando y unos directorios hijos en los que se almacenan aquellas funciones exclusivas de cada analizador, tal y como se aprecia en la *Figura 8*.

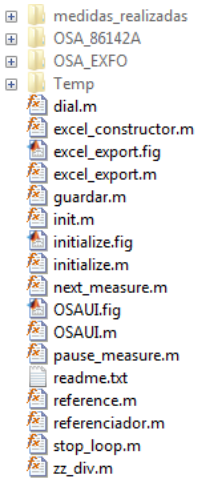

Figura 8. Directorio principal del software.

El código, como ya se ha establecido anteriormente, tiene una estructura de tipo árbol en la que a partir de una función principal (la interfaz gráfica) se desarrollan las distintas funciones dependientes de ésta. Debido a que Matlab se basa en la programación funcional, se ha descartado la programación orientada a objetos, aunque se podría optimizar el software y hacerlo mucho más eficiente con esta aproximación (ver sección *9. Líneas futuras*). Se hace necesario por ello la utilización de una variable global de tipo *struct* para poder intercambiar la información entre las distintas funciones de una forma ordenada [14], y se distingue entre dos bloques: el bloque de funciones comunes y el bloque de funciones propias de cada instrumento.

5.3.1.1 Diagrama de flujo principal.

El desarrollo de la interfaz gráfica de usuario parte del diagrama de flujo de la *Figura 9*. De esta forma, se amplían los distintos bloques de los que consta el programa en torno a los tres bloques especificados en el esquema general de la *Figura 7*.

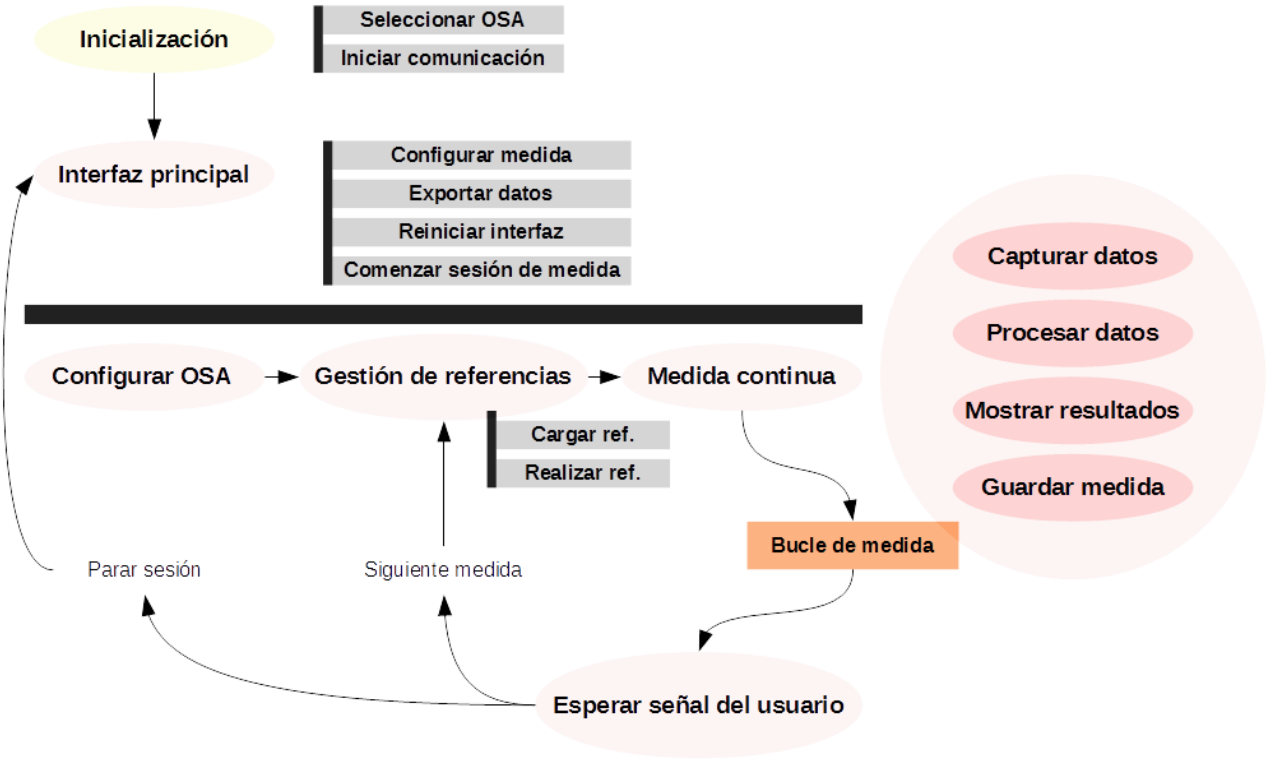

Figura 9. Diagrama de flujo principal del software.

5.3.1.2 Desglose de las distintas funcionalidades de la interfaz.

El funcionamiento general del software sigue el diagrama de flujo del apartado anterior. Cuando se inicia el programa, aparece una primera ventana de inicialización que permite al usuario elegir el OSA con el que se va a trabajar, cómo se van a gestionar las referencias, es decir, si se desean cargar medidas de referencia previamente guardadas o si se desea realizar un referenciado antes de cada nueva medida, y qué nivel de suavizado de las curvas se desea emplear. Los dos menús de configuración pueden observarse en las *Figuras 10* y *11* respectivamente.

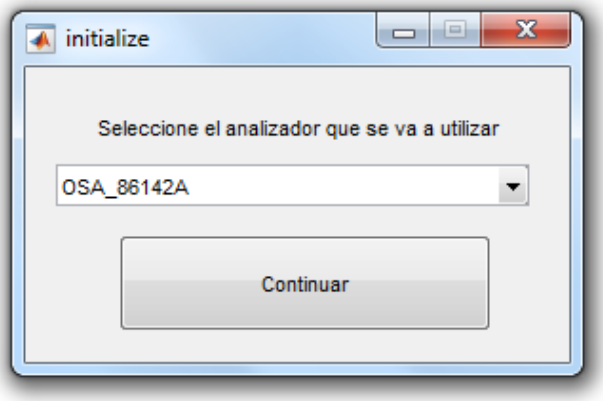

Figura 10. Selección modelo del analizador.

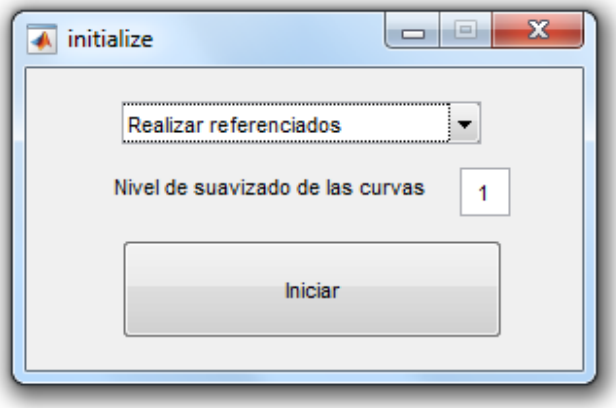

Figura 11. Gestión de referencias y suavizado.

Una vez especificado qué analizador se va a utilizar y cómo se van a tratar las referencias, el programa se conecta al instrumento y le envía las instrucciones de inicio correspondientes, que se encuentran detalladas en la sección *5.3.3.1. Función de inicialización del OSA.*

Cuando se ha realizado una conexión satisfactoria se inicia la ventana principal de la interfaz. En el caso de que haya alguna excepción en el intento de conexión e inicio del OSA, o si el instrumento no se encuentra operativo por estar apagado, aparecerá un cuadro de diálogo informando de dicha excepción. La ventana principal de la interfaz permite al usuario acceder a cuatro opciones como puede verse en la *Figura 12*.

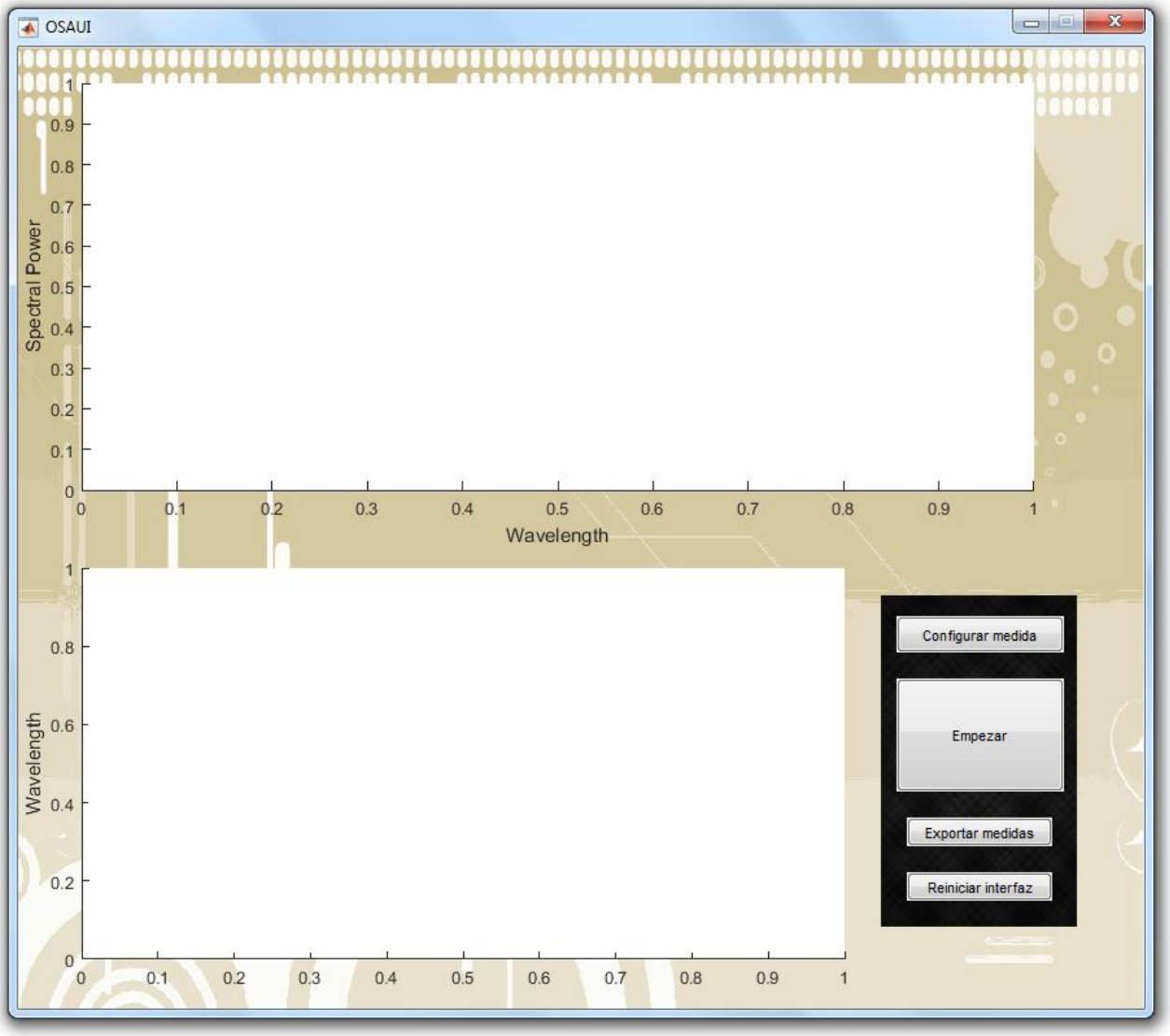

Figura 12. Ventana principal de la interfaz de usuario.

El botón de configuración de medida abre una ventana con opciones de configuración que dependerán del analizador que se esté utilizando. Es en esta ventana desde donde el usuario puede modificar los parámetros de configuración para programar el OSA para realizar la medida o medidas deseadas.

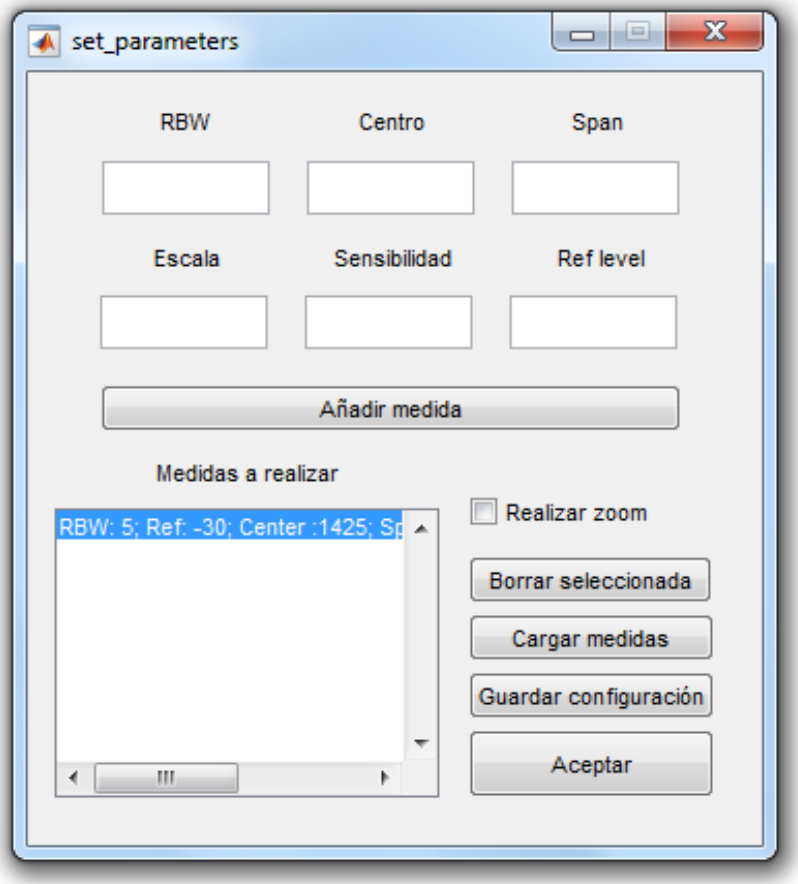

Figura 13. Ventana de configuración de parámetros para el OSA AGILENT 86142A.

La opción de reinicialización de la interfaz permite al usuario volver a establecer una comunicación con el instrumento si así lo desea.

El botón *Empezar* comienza todo el proceso de medición. Dependiendo del tipo de gestión de referencias que se haya seleccionado, al iniciar la interfaz se realizarán medidas de referencia o se preguntará al usuario cuál de las referencias previamente cargadas se desea utilizar. El control de la medición se realiza mediante una ventana externa que presenta tres opciones, *Parar*, *Siguiente* y *Pausa* (ver *Figura 14*). El botón de pausa interrumpe la medida continua hasta que el usuario decida volver a retomarla. El botón de siguiente medida avanza hacia la siguiente medida programada. En el caso de que se hayan programado varias configuraciones de medida distintas, el software configura el analizador, vuelve a cargar la referencia que se desee o realiza otro referenciado y se comienza el bucle de medidas nuevamente. Por último, el botón de parar la medida sale del bucle de medición, cierra la ventana de gestión de medidas y se devuelve el control a la ventana principal de la interfaz.

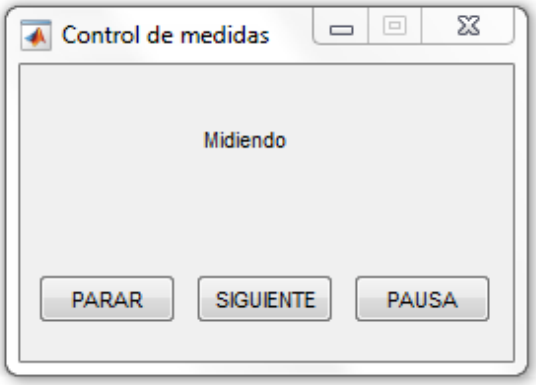

Figura 14. Dialogo control de medida.

Tal y como se aprecia en el diagrama de flujo expuesto en la *Figura 9*, el proceso de medida consta de tres pasos. En primer lugar, se debe configurar el analizador para la medida que se va a realizar. En segundo lugar, se debe tomar una referencia y en tercer lugar se comienzan a tomar medidas de forma continua y a mostrar los resultados en las ventanas de visualización integradas en la ventana principal de la interfaz. Cada vez que se realiza una medición, se almacenan los datos obtenidos en formato ".mat" dentro de un subdirectorio que se nombra con la fecha de la medida y contenido en el directorio "Medidas realizadas".

La opción de exportar datos, ejecuta una función preparada para exportar los datos obtenidos con el software dando opción al usuario de elegir el directorio desde donde se deben importar las medidas realizadas y permitiendo por ello exportar información de sesiones anteriores o exportar las medidas realizadas durante la sesión actual. El formato de salida de las medidas exportadas es ".xlsx" de Microsoft Office Excel.

5.3.1.3 Interacción con el usuario.

Tal y como se ha estructurado el software, hay dos tipos de interacción con el usuario. La interacción a través de la interfaz y la modificación del propio código. De esta forma, hay parámetros que no pueden ser configurados desde la interfaz gráfica pero que en caso de ser necesario pueden ser modificados editando el código.

La interacción a través de la interfaz se realiza mediante cuadros de dialogo, dejando la mayor parte de la ventana principal disponible para lograr una cómoda visualización de los datos obtenidos.

5.3.1.4 Estructura de los objetos gráficos y eventos de ejecución desarrollados.

Pueden definirse dos formas de trabajar dentro del entorno de desarrollo de Matlab para el diseño de una interfaz gráfica de usuario (*GUI – Graphical User Interface*). Una de estas formas es la creación de una GUI de forma programática, es decir definiendo todos y cada uno de los objetos gráficos y el control de ejecución de eventos desde cero, lo cual permite alcanzar el máximo nivel de personalización posible [15]. La otra forma es utilizar *guide*, un entorno de diseño de ventanas gráficas, implementado dentro de Matlab y que genera una figura que se controla desde el archivo de código correspondiente [16].

Debido a la relativa complejidad de la interfaz que se ha desarrollado, se ha optado por combinar ambas formas de trabajo, siendo la utilización de la maquetación gráfica con *guide* la herramienta fundamental para el diseño tanto de la ventana principal como de las distintas ventanas de configuración a las que se accede desde la interfaz. Los cuadros de dialogo, por otro lado, se han creado de forma programática.

Las razones fundamentales por las que se ha optado por utilizar *guide* en las ventanas de mayor complejidad son la facilidad de crear objetos gráficos relativamente complejos de programar desde cero (como *pop-ups* de selección múltiple, *check-boxes* o la presencia de distintos menús que varíen de forma dinámica dentro de una misma ventana) y las opciones de organización y alineación de elementos que ofrece el entorno proporcionado por esta herramienta.

Por el contrario, aquellos diálogos creados para comunicar información al usuario o para controlar la ejecución del programa, se han diseñado de forma programática para reducir tanto el consumo de recursos del software como el número de archivos en el directorio raíz de la aplicación, así como aprovechar la potencia de personalización que ofrece esta aproximación.

Las dos formas de desarrollo expuestas, operan de la misma manera, siendo la diferencia fundamental entre ellas la automatización que realiza la herramienta *guide* al generar el código

mínimo necesario para la creación de los objetos gráficos de forma automática, siendo éste código completamente personalizable por el desarrollador.

La estructura del código de las ventanas creadas a partir de *guide* se basa en el manejo de *callbacks,* o llamadas a funciones definidas en este archivo y que se asocian a las distintas acciones que puede realizar el usuario en la interfaz. De esta forma, cuando el usuario realiza una acción interactuando con la interfaz, se ejecuta el *callback* asociado a dicha acción y el código que se haya incluido en dicho *callback* [17]. También se genera una estructura de variables propia de la función, nombrada por defecto *handles* en la que se almacenan todas las propiedades de los objetos gráficos de los que está formada la interfaz y que permite al programa controlar el estado de cada uno de estos objetos y la interacción del usuario con dichos objetos [18].

El formato estructural del archivo de código de una interfaz gráfica de usuario consiste en la definición de una función principal que da nombre al archivo, de una función de inicialización que se ejecuta antes de que se genere la figura que contiene la interfaz, una función de salida que retorna los *handles* cuando se produce una acción o modificación de los valores de dichas variables y los *callbacks* que haya definido el usuario para controlar la ejecución de la interfaz.

5.3.1.5 Estructura de variables principal y definición dinámica de la misma.

Se procederá a describir la estructura de variables tipo *struct* [14] de Matlab de la que hace uso el software desarrollado para el control y correcto funcionamiento del mismo. La variable principal y que facilita el intercambio de información entre las distintas funciones de las que está compuesto el programa es la estructura de variables global “data”. Se ha optado por una variable global para no aumentar la complejidad del flujo de información entre las funciones y poder manejar la organización estructural del programa con mayor libertad, sin la necesidad de encadenar las funciones (las salidas de unas funciones con las entradas de las siguientes en el orden de ejecución). Se consiguen, por ello, funciones independientes, que pueden ser llamadas en distinto orden sin necesidad de definir múltiples entradas y salidas que variarían según el flujo de ejecución elegido por el usuario al operar con la interfaz gráfica. De esta forma, las funciones van añadiendo y borrando datos de la estructura global de forma dinámica y en el orden en el que se van necesitando.

Un ejemplo claro de esta definición de variables de forma dinámica está en el almacenamiento de las todas las medidas que se realizan desde que se comienza a medir hasta que el usuario para el bucle de obtención de datos continuo, datos que se van añadiendo a la estructura "data" (ver *Fragmento de código 11*, en la sección *5.3.2.2. Ventana principal de la interfaz*)

Dentro de esta estructura global se distinguen tres tipos de variable:

- Variables de control del flujo de programa. Este tipo de variables almacenan valores que permiten al software decidir el orden de ejecución de las distintas funciones, definir el camino seguido dentro del flujo y controlar las iteraciones que se realizan en la gestión de referencias, medición, visualización de los datos y guardado de medidas.

- Variables de configuración. Son variables que toman valores que especifican los parámetros elegidos por el usuario tanto para para la configuración del analizador, como para la presentación de los datos obtenidos.

- Variables de almacenamiento de datos. Estas variables almacenan los datos obtenidos en las medidas realizadas, para poder procesar esta información, operar con ella y finalmente mostrarla por pantalla.

Dentro de todas las variables que maneja el programa, las hay también constantes, es decir, que su valor está predefinido y no varía en ningún momento durante la ejecución. Estas variables definen aspectos de control del software avanzados y únicamente pueden ser modificadas editando el código (ver anexos, sección: *11.2 Desglose de la estructura de variables principal "data").*

5.3.2 Funciones comunes

Esta sección, desglosa las distintas funciones de las que se sirve el programa para su funcionamiento y que son comunes a cualquier instrumento que se desee controlar.

5.3.2.1 Inicialización del software.

La primera función que se ejecuta cuando se inicia la interfaz es la función "initialize.m". Este archivo define una pequeña ventana de diálogo en la que el usuario debe seleccionar el instrumento que se va a utilizar, dentro de los instrumentos para los que se haya configurado la interfaz. Una vez seleccionado el modelo de OSA aparece la segunda opción de configuración dentro de esta ventana, que obliga al usuario a elegir como se van a tomar las referencias y si desea cargar unas referencias previamente guardadas, así como prefijar el nivel de suavizado de curva que se desea en las medidas (ver sección *5.3.3.6 Procesado de los datos*).

Como se aprecia en el *Fragmento de código 3,* esta función define e inicializa las variables "data.path", "data.primary_path", "data.osa" y "data.ref_control" que almacenan la información de la ruta de localización de las funciones propias del analizador elegido, la ruta del directorio que contiene las funciones comunes, el modelo de OSA utilizado y la forma de tratamiento de las referencias respectivamente.

Por último, ejecuta la función de inicio del analizador seleccionado dentro del directorio correspondiente realizando una llamada dinámica a la función. Para ello, como puede apreciarse en el *Fragmento de código 1*, en primer lugar, se genera el nombre de la función a partir del nombre del modelo seleccionado y que se encuentra almacenado en la variable "data.osa" y se guarda en la variable temporal "osa", y en segundo lugar simplemente se hace una llamada a esta función generada.

Fragmento de código [1]. Llamada a función generada de forma dinámica.

```
osa=str2func(sprintf('%s_init',data.osa));
osa();
```

La ventana de inicialización consta de dos menús distintos, uno para cada una de las dos configuraciones que el usuario debe realizar antes de iniciar la interfaz. La primera de ellas consiste en la selección del OSA que se va a utilizar. Para ello se ha programado un *pop-up* de selección desplegable en el que el usuario debe seleccionar el modelo de analizador y un botón de aceptar. Por esta razón, en el código que configura la ventana cuando esta se genera y se muestra al usuario (función "initialize_OpeningFnc", *Fragmento de código 2*), se ocultan los objetos gráficos correspondientes a la segunda configuración.

Una vez realizada esta selección, el *pop-up* desaparece y se hacen visibles los objetos ocultados durante la inicialización de la ventana. Así este *pop-up* es sustituido por otro en el que el usuario debe realizar la selección de gestión de referencias y que se configura con un *callback* distinto. También se hace visible un cuadro de edición de texto en el que el usuario debe introducir el nivel de suavizado de las curvas y que es controlado por el *callback* correspondiente. El botón de aceptar utiliza la misma función de *callback* para ambos casos, avanzando de uno a otro a partir de la comprobación del *string* del propio botón como puede apreciarse en el *Fragmento de código 3* en el que se detalla la función de callback para el botón de aceptar de dicha ventana de inicialización.

Fragmento de código [2]. Función de inicialización de la ventana "initialize.m".

```
function initialize_OpeningFcn(hObject, eventdata, handles, varargin)
global data;
data.zoom=0;
data.error='';
set(handles.popup_referencia,'visible','off');
set(handles.text_smooth,'visible','off');
set(handles.input_smooth,'visible','off');
set(handles.button_initialize,'string','Continuar');
handles.output = hObject;
guidata(hObject, handles);
```

Fragmento de código [3]. "Callback" para el botón de aceptar del menú de inicialización.

```
function button_initialize_Callback(hObject, eventdata, handles)
global data
if strcmp(get(handles.button_initialize,'string'),'Continuar')
```

```
    w=what;
    data.primary_path=w.path;
    osa=get(handles.popup_osa,'value');
    switch osa
        case 1
            data.path=[data.primary_path '\OSA_86142A'];
            data.osa='AGILENT';
        case 2
            data.path=[data.primary_path '\OSA_EXFO'];
            data.osa='EXFO';
        otherwise
            return
    end
    set(handles.button_initialize,'string','Iniciar');
    set(handles.text_osa,'visible','off');
    set(handles.popup_osa,'visible','off');
    set(handles.popup_referencia,'visible','on');
    set(handles.text_smooth,'visible','on');
    set(handles.input_smooth,'visible','on');
    set(handles.input_smooth,'string','1');
else
    data.smooth=str2double(get(handles.input_smooth,'string'));
    if get(handles.popup_referencia,'value')==1
        data.ref_control=1;
    else
        data.ref_control=0;
    end
    q = instrfind;
    if length(q) ~= 0
        fclose(q);
    end
    delete(q);
    delete(gcf);
    reference();
    cd (data.path)
    try
        osa=str2func(sprintf('%s_init',data.osa));
        osa();
    catch
        cd (data.primary_path);
        msg=sprintf...
```

```
('No se ha podido inicializar la comunicación con el osa %s.. Revise que el
aparato esté conectado y configurado',data.osa);
        dial(msg);
        cd (data.path);
    end
cd (data.primary_path)
OSAUI();
end
```

Una vez el usuario supera los dos menús de configuración, se ejecuta la función "OSAUI.m" que contiene el código principal y define y muestra la ventana principal de la interfaz.

5.3.2.2 Ventana principal de la interfaz.

La ventana principal de la interfaz se encuentra definida en el archivo "OSAUI.m", que podría considerarse el archivo principal del software y cuyo nombre se debe al acrónimo de *Optical Spectrum Analyzer User Interface*. Este *script* es el conjunto de funciones que definen todos los aspectos de esta ventana y los relacionan con la figura generada por la herramienta *guide*. Contiene también todos los *callbacks* necesarios para que los objetos gráficos sean operativos. Así, consta de cuatro botones que corresponden a las operaciones de: configuración de medidas, iniciar mediciones, exportar medidas y reinicialización de la interfaz. También cuenta con dos ejes de coordenadas mediante los cuales se mostrarán los datos al usuario.

La generación de la figura que contendrá la ventana principal se realiza, como cualquier otra ventana del software, mediante la definición de tres funciones. La primera de ellas, define el archivo y la estructura, la segunda contiene el código de inicialización y configuración de la ventana y la tercera de ellas la envía a la pantalla. Todas ellas se ejecutan antes de que aparezca el objeto gráfico. Como se puede apreciar en el *Fragmento de código 4*, se desactivan todos los avisos por la consola de Matlab durante la inicialización de la ventana principal.

Si se analiza detenidamente el código de inicialización de esta ventana, se puede apreciar cómo gran parte del código está destinado a utilizar dos imágenes guardadas en el directorio oculto "~/temp" como fondo de la ventana con un objetivo puramente estético. Para esto, se hace uso de

dos ejes de coordenadas definidos previamente en la figura creada mediante *guide*, que cubren toda la superficie de la ventana y el espacio que rodea a los botones respectivamente.

Fragmento de código [4]. Función de inicialización de "OSAUI.m".

```
function OSAUI_OpeningFcn(hObject, eventdata, handles, varargin) %#ok<*INUSL>
global data; warning off all;
[stat,struc] = fileattrib;   %#ok<ASGLU>
PathCurrent = struc.Name;   %obtiene la direccion del directorio raiz
axes(handles.fondo_principal);
a=imread([PathCurrent '/Temp/wall.jpg']);
image(a); axis off;
axes(handles.fondo_controles);
a=imread([PathCurrent '/Temp/grid.jpg']);
image(a); axis off;
axes(handles.ejes_medida);
axes(handles.ejes_pajaro);
set(handles.button_start,'enable','off');
ylabel(handles.ejes_medida,'Spectral power')
xlabel(handles.ejes_medida,'Wavelength')
ylabel(handles.ejes_pajaro,'Wavelength')
data.last=0;
handles.output = hObject;
guidata(hObject, handles);
```

A continuación, se procederá a describir el código ejecutado al ser presionados cada uno de los botones de la interfaz.

El botón de configurar medidas, como se puede apreciar en el *Fragmento de código 5,* navega hasta el sub-directorio correspondiente al analizador seleccionado y ejecuta la función que define su ventana de configuración (archivo "set_parameters.m").

Fragmento de código [5]. Callback para el botón de configuración de medidas.

```
function button_config_Callback(hObject, eventdata, handles)
```

```
global data
cd (data.path)
try
    set_parameters();
catch
    delete(gcf)
    cd (data.primary_path);
    msg='Ejecute primero la función de inicialización';
    dial(msg)
    cd (data.path);
end
```

El botón de exportar medidas, análogamente al de configuración, ejecuta el archivo "excel_export.m" que contiene las rutinas necesarias para exportar las medidas al formato de Microsoft Office ".xlsx".

Se considera necesario, por otro lado, hacer una referencia al *Fragmento de código 6* ya que, en el *callback* correspondiente al botón de reinicio de la interfaz, es necesario cerrar la ventana principal, borrar de la memoria todas las variables que se hayan inicializado, asegurar que se vuelve al directorio principal del programa y ejecutar la función "init.m" para iniciar el software de nuevo.

Fragmento de código [6]. Callback para el botón de reinicialización de la interfaz.

```
function button_reinitialize_Callback(hObject, eventdata, handles)
global data;
cd (data.primary_path);
delete (handles.figure1);
try
    clear all;
catch
    return;
end
init();
```

Además de las funciones de *callback* de los botones, el código de la ventana principal precisa de un *callback* para cerrar adecuadamente la figura ante una petición de cierre de la ventana por parte del usuario. Ésta función de *callback* se declara utilizando la palabra reservada de Matlab "CloseRequestFcn", tal y como se observa en el *Fragmento de código 7.*

Fragmento de código [7]. Callback para la petición de cierre de ventana.

```
function figure1_CloseRequestFcn(hObject, eventdata, handles)
try
    global data;
    cd (data.primary_path);
    delete(hObject);
    clear all;
catch
    delete(gcf);
end
```

Por último, se analizará la función de *callback,* más importante de la ventana principal, que contiene el código que debe ejecutarse cuando el usuario pulsa el botón "comenzar" (*Figura 15*).

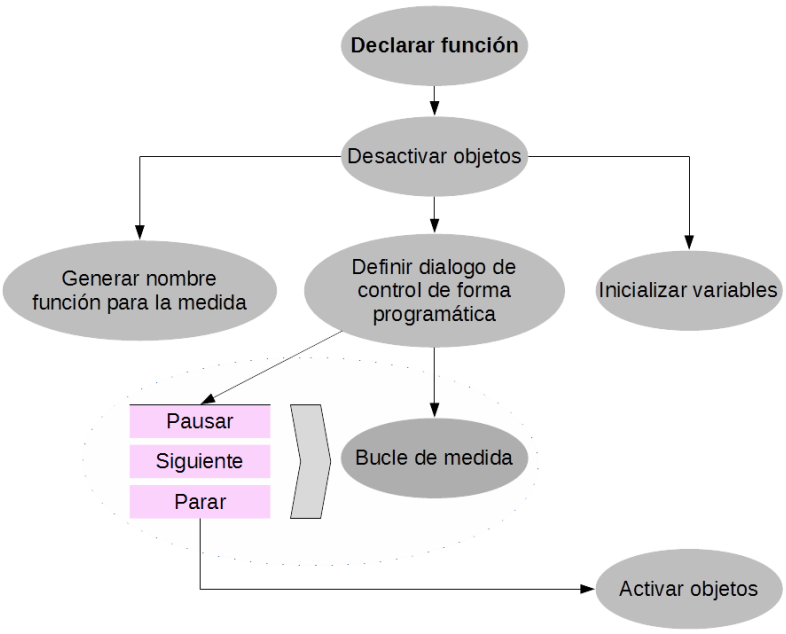

Figura 15. Diagrama de flujo para el callback del botón "comenzar".

Este *callback* sigue el diagrama de flujo de la *Figura 15*. Cuando se pulsa el botón de comenzar medida, tras declarar la estructura de variables global "data" para utilizarla a lo largo del *callback*, se desactivan los botones de la interfaz para impedir que el usuario pueda dar cualquier orden que entre en conflicto con la obtención de datos y que resultaría en una excepción en la ejecución del código (*Fragmento de código 8*). Un ejemplo claro de dicha excepción sería que el programa recibiese la orden de configurar una medida mientras se está realizando el proceso de medida.

Fragmento de código [8]. Desactivación de objetos gráficos.

```
set(handles.button_config,'enable','off');
set(handles.button_start,'enable','off');
set(handles.button_export,'enable','off');
```

Una vez que se han desactivado los botones de acción de la interfaz, se ejecutan las tres rutinas de inicialización necesarias antes de entrar en el bucle de medida, la definición de un cuadro de diálogo con el que el usuario pueda interactuar, la inicialización de las variables que precisen de un valor inicial antes de entrar en el bucle de medida y la generación del nombre de la función que gestionará las medidas en el analizador y cuyo nombre dependerá del modelo de OSA seleccionado en la configuración inicial.

En primer lugar, se define el cuadro de control de la medida de forma programática. Como puede verse en el *Fragmento de código 9*, se definen los tres botones de acción "parar", "siguiente" y "pausa".

El código que se ejecuta cuando se pulsa uno de estos botones presentes en el diálogo se encuentra definido en las tres funciones de control de medida "stop_loop.m", "next_measure.m" y "pause_measure.m" que modificarán las variables de control necesarias para realizar las acciones correspondientes (ver sección *5.3.2.7 Funciones de control de medida*). Para asegurar la ejecución de estas funciones independientemente de la ruta en la que esté trabajando el software, es necesario cargar la estructura de variables global "data" y hacer uso del comando "cd (data.primary_path)" para navegar hasta el directorio en el que se encuentran dichas funciones.

Este cuadro de control únicamente se cerrará si el usuario presiona el botón parar, mediante el comando “close (gcf)” o en el caso de que se produzca una excepción en la ejecución del código.

Fragmento de código [9]. Definición del cuadro de dialogo de control de medida.

```
msg='Midiendo';
    d = dialog('Position',[500 400 250 150],'Name','Control de medidas');
    txt = uicontrol('Parent',d,...
              'Style','text',...
              'Position',[10 80 210 40],...
              'String',msg); %#ok<*NASGU>

    btn = uicontrol('Parent',d,...
              'Position',[10 20 70 25],...
              'String','PARAR',...
              'Callback','delete(gcf);global data;...
               cd(data.primary_path);stop_loop;cd(data.path)');
    btn2 = uicontrol('Parent',d,...
              'Position',[90 20 70 25],...
              'String','SIGUIENTE',...
              'Callback','global data;...
               cd(data.primary_path);next_measure;cd(data.path)');
    btn3 = uicontrol('Parent',d,...
              'Position',[170 20 70 25],...
              'String','PAUSA','Callback','global data;...
               cd(data.primary_path);pause_measure;cd(data.path);');
```

En segundo lugar, y una vez se ha generado el diálogo de control, la función de *callback* del botón de “comenzar medida” inicializa las variables de control de flujo, como se ve en el *Fragmento de código 10,* y se genera el nombre de la función de gestión de medidas que se llamará durante el bucle de medidas con el alias “osa” (*Fragmento de código 11).* La generación de un alias se realiza mediante la instrucción “str2func” que sigue la sintaxis:

str = ‘function_name’; fh = str2func(str);

Donde *str* es la cadena de caracteres que contiene el nombre de la función de la que se desea crear un alias y *fh* es el alias creado. De esta forma *fh* se convierte en una función anónima [19].

Fragmento de código [10]. Inicialización de variables antes del bucle de medida.

```
data.A = 0;
data.zoom_level=2;
data.loop=0;
data.control=1;
data.cont=1;
l=data.last;
```

Fragmento de código [11]. Generación de alias para la llamada a la función principal de gestión del analizador.

```
osa=str2func(sprintf('%s_start',data.osa));
```

Las variables que se definen en el *Fragmento de código 10,* controlan el bucle de medidas. La variable “data.A” almacenará en forma de matriz “n x m”, donde “n” es el número de filas y por tanto el número de puntos obtenidos y “m” el número de columnas o de medidas de potencia espectral realizadas más el vector de longitudes de onda con los datos correspondientes a las medidas realizadas desde que el usuario pulsa el botón comenzar hasta que detiene la medida. La variable “data.loop” indica si se ha realizado la configuración del analizador para realizar la medida. Si su valor es “0”, implica que el OSA aún no se ha configurado, es decir, que el bucle de medidas se encuentra en la primera iteración. Por ello se inicializa en el valor “0” antes de entrar en el bucle de medidas, con el objetivo de garantizar la configuración adecuada del analizador antes de la primera solicitud de datos. Una vez configurado el analizador, y hasta que el usuario pulse uno de los botones de control del diálogo de control, esta variable tomará el valor 1, indicando que el bucle de medidas puede operar sin realizar ningún tipo de configuración en el OSA. La variable “data.control”, es la variable de la que hace uso el bucle de medidas como condición para continuar iterando. Se inicializa en el valor “True” para comenzar a medir, y toma el valor “False” cuando el usuario realiza una petición de interrupción del bucle de medidas.

Dentro del control de las medidas que se van a realizar, y ante la posibilidad de configurar varias medidas distintas en una misma sesión, se hace necesario disponer de una variable que indique al software qué medida se va a realizar. Esta variable es “data.cont”, un contador que se va incrementando en una unidad por cada medida con una configuración distinta que se realiza. La función de control de medida “next_measure.m” hace uso de esta variable para cargar los parámetros de configuración correspondientes a la siguiente medida programada. Se inicializa en el valor “1” y puede incrementarse hasta el máximo determinado por la variable “data.last” que almacena el número de medidas añadidas por el usuario en la ventana de configuración.

Además de estas tres variables, que tienen un ámbito de trabajo global ya que son manipuladas por funciones auxiliares a la ventana principal, se inicializa la variable “c” con el valor “1”. Esta variable es un contador que almacena el número de iteraciones que se han realizado con una misma configuración de medida y permite al código realizar un correcto manejo de la instrucción *pcolor* dentro del bucle de medidas tal y como puede observarse en el *Fragmento de código 12*.

El bucle de medidas, como ya se ha adelantado en la descripción de las variables que precisan inicializarse con anterioridad a su ejecución, consiste en la evaluación de la variable “data.control”, mediante una instrucción condicional de tipo *while*.

La parte más importante del código de este bucle llama a la función de gestión de medidas, mediante el alias generado anteriormente (“osa”), enviándole como argumento la variable “data.loop”. La función de gestión, devolverá los dos vectores con los datos que deben mostrarse en el eje de coordenadas de situado en la parte superior de la ventana principal de la interfaz, en la que se representa la potencia espectral frente a las longitudes de onda tal y como se explica en la sección *5.3.3.5 Función principal de gestión del analizador*.

A continuación, se añade el vector de potencia espectral devuelto por la función de gestión de medidas “y_plot” a la variable “data.A” en la columna indicada por el contador de iteraciones realizadas “c”. El argumento condicional “if data.loop == 0” comprueba si se trata de la primera iteración del bucle de medidas. Si se cumple esta condición, almacena los datos obtenidos en las variables “data.x_persistent” y “data.y_persistent” que se mantendrán fijas en el eje de coordenadas durante el resto de iteraciones.

Si se ha realizado más de una iteración del bucle, además de actualizar el eje de coordenadas de potencia espectral frente a longitudes de onda, se ejecuta la instrucción *pcolor* que dibuja un mapa de colores en el segundo eje de coordenadas, situado en la parte inferior de la ventana principal de la interfaz.

Puede resultar llamativo que la primera instrucción del bucle sea "pause(0.01)". Esta instrucción interrumpe la ejecución del programa durante una décima de segundo y, aunque pueda parecer redundante, esta pausa es necesaria para que el intérprete de código de Matlab pueda evaluar la condición del bucle *while* correctamente e interrumpirlo si se recibe una petición de ello por parte del usuario, ya que únicamente se hace uso de un hilo de ejecución.

De forma ideal, se crearía un hilo de ejecución exclusivo para el bucle de medida. Así, se ejecutaría el bucle iterativo en segundo plano, manteniendo el hilo principal exclusivamente para el control de la interfaz. De esta forma, el cuadro de control de medida sería redundante, ya que cambiando las cadenas de caracteres de los botones e implementando un condicional en sus funciones de *callback* se permitiría al usuario realizar las mismas acciones con mayor comodidad. Las dificultades que implica esta aproximación mediante Matlab han determinado que se utilice un único hilo de ejecución y que se precise de dicho cuadro de control de medida. Para que el usuario pueda interactuar con la interfaz mientras el bucle *while* se encuentra activo, se simula un segundo hilo de ejecución mediante llamadas a funciones externas a la función principal y una pausa en el bucle de medidas. Esto permite a Matlab ejecutar las funciones asociadas a los botones del cuadro de diálogo de control de medida si así lo requiere el usuario sin dejar de evaluar la variable condicional del bucle de medidas; todo ello dentro del mismo hilo de ejecución. Aunque se ha considerado que el denominado *Multi-threading* está fuera de los objetivos de este trabajo, se ha propuesto como línea futura (ver sección *8. Conclusiones* y *9. Líneas futuras*).

Fragmento de código [12]. Bucle de medidas.

```
while data.control                %Bucle de medidas
pause(0.01);
cd (data.path);
    try
        [x, y_plot]=osa(data.loop);
        axes(handles.ejes_medida); %#ok<*LAXES>
```

```
            if data.loop==0
                data.loop=1;
                data.x_persistent=x; data.y_persistent=y_plot;
                data.A=x;
                data.A(:,2)=y_plot;
                cd (data.primary_path)
                guardar(data.param,1,data.zoom,1)
                cd (data.path)
                plot(x,y_plot,'g.');
            else
                s = size(data.A);
                data.A(:,s(2)+1)=y_plot;
                cd (data.primary_path)
                guardar(data.param,0,data.zoom,0)
                cd (data.path)
                plot(data.x_persistent,data.y_persistent,'g.',x,y_plot,'r.');
                axes(handles.ejes_pajaro);
                        pcolor(1:c,x,data.A(:,2:(s(2)+1)));
                shading interp;
                drawnow;
            end
            c = c+1;

        catch
            try
                cd (data.primary_path);
                stop_loop;
                msg=['Error: ' data.error ' //Se reiniciará la interfaz//'];
                dial(msg);
            catch
                return
            end
            close all;
            clear all; %#ok<*CLALL>
            init();
            return
        end
    cd (data.primary_path);
    end
```

Una vez que el usuario realiza una petición de interrupción del bucle de medidas, la función de *callback* del botón de empezar medidas vuelve a activar el resto de botones de la ventana principal de la interfaz.

5.3.2.3 Función de dialogo.

Dentro de las funciones de uso común que se hallan almacenadas en el directorio principal del software, se ha desarrollado un cuadro de dialogo de uso general que se encuentra definido de forma programática en la función "dial.m" y que ya ha aparecido en los fragmentos de código citados a lo largo del documento. Como puede observarse en el *Fragmento de código 13*, esta función, a partir del argumento de tipo *string*, conteniendo la frase que se desee mostrar al usuario, genera un cuadro de diálogo con un botón de "ok" e interrumpe la ejecución de código hasta que este botón sea presionado. La interrupción de la ejecución del código se realiza mediante la instrucción *uiwait(d)* donde "d" es el objeto de tipo *dialog* que se genera en la función "dial.m" [20]. De esta forma, Matlab espera hasta que el objeto gráfico "d" es destruido, lo cual únicamente ocurre si el usuario presiona el botón de "ok", mediante la instrucción *delete(gcf)*.

Ésta función se ha utilizado reiteradamente a lo largo de todo el software desarrollado para enviar mensajes de carácter informativo al usuario, sin la necesidad de generar un cuadro de dialogo de forma programática cada vez que se precisa de ello.

Para realizar una llamada a esta función desde cualquier otra función o parte del código de la interfaz, basta con utilizar la instrucción "cd (data.primary_path)" para dirigir al intérprete al directorio principal del software y ejecutar desde ahí la llamada "dial(msg)", donde el argumento *msg* es la variable que contiene el *string* que se desea comunicar al usuario.

Fragmento de código [13]. Función "dial.m".

```
function dial(msg)
    d = dialog('Position',[300 300 250 150],'Name','Información');
    txt = uicontrol('Parent',d,...
               'Style','text',...
```

```
                'Position',[20 80 210 40],...
                'String',msg);

    btn = uicontrol('Parent',d,...
                'Position',[85 20 70 25],...
                'String','OK',...
                'Callback','delete(gcf)');
    uiwait(d);
end
```

5.3.2.4 Función de cargado de referencias.

La función "reference.m" es la que se encarga de generar los cuadros de diálogo necesarios para permitir al usuario cargar archivos con medidas de referencia ya realizadas y guardadas previamente a la inicialización del software, extraer la información contenida en dichos archivos y almacenarla en las variables correspondientes para su posterior uso durante la operación de tratamiento de datos.

Esta función es llamada únicamente desde la función de inicialización ("initialize.m") cuando el usuario debe escoger cómo se van a gestionar las referencias. Así, como puede verse en el *Fragmento de código 14*, si el usuario ha escogido la opción de realizar un referenciado antes de cada medida, la función retorna sin ejecutarse. Si, por el contrario, mediante la evaluación de la variable "data.ref_control" es necesario cargar referencias, comienzan a ejecutarse las rutinas que generarán los cuadros de diálogo necesarios para permitir al usuario navegar hasta los archivos que contienen las medidas de referencia.

Mediante el uso de la instrucción *[file, path] = uigetfile('*.xlsx')*, se abre el explorador de Windows en el directorio principal del software (que es el directorio desde donde el software está ejecutando la función "reference.m") con la posibilidad de seleccionar archivos con extensión ".xlsx" [21]. Esta es la extensión en la que se guardan, si el usuario así lo desea, los datos obtenidos como referencia, cuando el software se configura para realizar referenciados antes de cada medida. La función permite al usuario cargar hasta un máximo de veinte referencias.

Los datos de estos archivos se extraen a partir de la instrucción de Matlab *xlsread* y, en caso de necesitar almacenar más de una referencia, se emplea una declaración dinámica de variables mediante el uso de un bucle de tipo *for*, realizando tantas iteraciones como referencias se deban cargar y realizando una asignación de los nombres de las variables mediante la instrucción *sprintf*:

```
data.(sprintf('x_reference%d',i))=A(1:length(A),1);
data.(sprintf('y_reference%d',i))=A(1:length(A),2);
```

Donde en las instrucciones anteriores, "data" es la estructura de variables global a la que se van a añadir dos campos por cada referencia cargada, uno conteniendo el vector de longitudes de onda y otro conteniendo el vector de potencia espectral. La variable "x_reference" es el vector de longitudes de onda e "y_reference" el de potencia espectral, el parámetro "i" es el contador de iteraciones del bucle *for* y "A" es la variable temporal en la que se han cargado los datos contenidos en el fichero de tipo ".xlsx" mediante la instrucción *xlsread*.

Fragmento de código [14]. Función "reference.m".

```
function reference()
global data
if data.ref_control==1
    return
else
    choice = questdlg('¿Cuantas referencias son necesarias?', ...
      'Es cuestion de cantidad', ...
      'Una','Más de una','Cancelar');
    switch choice
        case 'Una'
            [file,path] = uigetfile('*.xlsx');
            if path~=0
                data.ref_num=1;
                A=xlsread([path file]);
                data.x_reference=A(1:length(A),1);
                data.y_reference1=A(1:length(A),2);
                data.puntos=length(data.x_reference);
                msg='Se ha cargado la referencia';
                dial(msg);
                delete(gcf);
            else
                return;
```

```
        end
    case 'Más de una'
        prompt = {'¿Cuantas?:'};
        dlg_title = 'Ref';
        num_lines = 1;
        defaultans = {''};
        answer = inputdlg(prompt,dlg_title,num_lines,defaultans);
        data.ref_num=str2double(answer{1});
        a=data.ref_num;
        if ~isnan(a) && a<20
            for i=drange(1:1:a)
                [file,path] = uigetfile('*.xlsx',...
                (sprintf('Seleccione referencia num. %d',i)));
                if path~=0
                    A=xlsread([path file]);
                    data.(sprintf('x_reference%d',i))=A(1:length(A),1);
                    data.(sprintf('y_reference%d',i))=A(1:length(A),2);
                else
                    return;
                end
            end
            msg='Se han cargado las referencias'; dial(msg);
            return
        else
            msg='El máximo de referencias admitido es de 20';
            dial(msg);
        end
    case 'Cancelar'
        delete(gcf);
end
```

5.3.2.5 Función de gestión de referencias.

Su cometido es encargarse de suministrar a la función de gestión de medidas y tratamiento de datos de cada OSA las referencias para compararlas con la medida realizada cada vez que el software lo requiere. Se encuentra definida en el archivo “referenciador.m” y su diagrama de flujo puede resumirse en el esquema de la *Figura 16*.

La llamada a esta función se realiza desde la rutina principal de gestión de medidas de cada analizador y que puede consultarse en la sección *5.3.3.5. Función principal de gestión del analizador*.

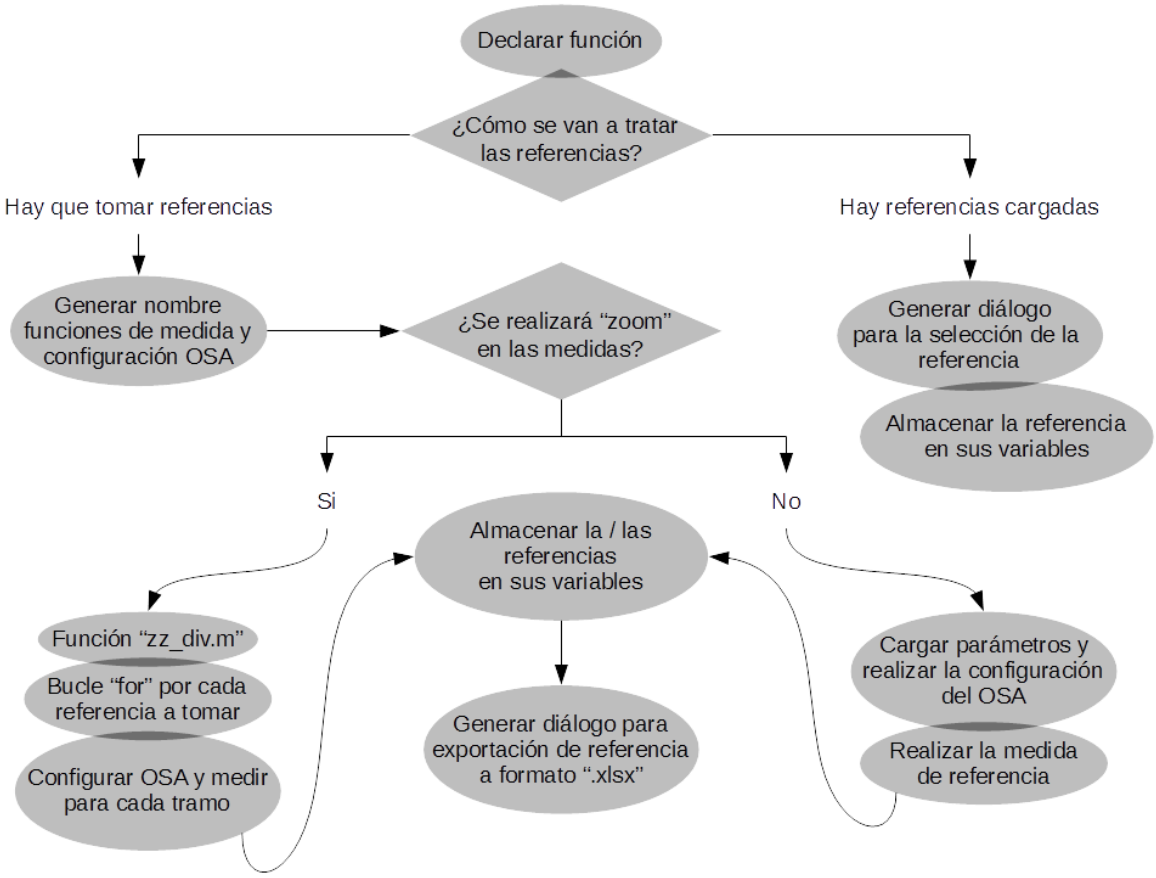

Figura 16. Diagrama de flujo de la función "referenciador.m"

La función acepta cuatro argumentos de entrada: "obj", "zoom", "center" y "span". El argumento "obj", es el objeto de comunicación que se genera durante la inicialización del instrumento y que es necesario pasar a las funciones tanto de configuración como de medida del analizador (ver secciones *5.3.3.1. Función de inicialización del OSA*, *5.3.3.2. Función de configuración del OSA* y *5.3.3.3. Función de medida del OSA*). El argumento "zoom" indica si el usuario ha solicitado realizar varias medidas dentro del rango configurado, dividiendo dicho rango en los tramos especificados. Por último, los argumentos "center" y "span" definen el rango en el que se va a realizar la referencia.

Como ya se ha expuesto en este documento, las referencias pueden haber sido cargadas previamente o tomarse antes de iniciar el bucle de medidas. Así, la primera instrucción de la función

"referenciador.m", tras cargar la estructura de variables global, comprueba cuál de las dos opciones se ha seleccionado evaluando la variable "data.ref_control".

En el caso de que haya referencias cargadas en memoria, el usuario debe elegir cuál de éstas se va a utilizar. Para ello se combina la generación de un dialogo de forma programática que permita al usuario introducir el número de la referencia que desea cargar con el uso de la función "dial.m" (ver *Fragmento de código 15*).

Fragmento de código [15]. Generación del diálogo para la selección de referencia cargada.

```
    k=data.ref_num;
    if k==1
        data.ref=data.y_reference1;
        data.x=data.x_reference;
    else
        cond=true;
        while cond
            try
                dlg = inputdlg(['Hay cargadas ' num2str(k) ' referencias, ¿cuál
desea utilizar?'],'Sample');
                input = str2double(dlg{:});
                data.ref=data.(sprintf('y_reference%d',input));
                data.x=data.(sprintf('x_reference%d',input));
                cond=false;
            catch
                msg='ERROR: Elija una referencia que se haya cargado..';
                dial(msg);
            end
        end
    end
```

Por otro lado, si es necesario realizar referenciados, se hace una distinción mediante la evaluación de la variable "data.zoom".

La salida que produce la función es variable únicamente si se han configurado las medidas para realizar un zoom. En el caso de no realizar un zoom, la salida de la función es idéntica tanto si se han cargado medidas de referencia como si se realiza un referenciado generando dos vectores que contienen las longitudes de onda en las que se ha medido y la potencia espectral contra la que se compararán las medidas que se van a efectuar posteriormente. Los dos vectores de salida que se producen (salida indirecta a través del uso de la estructura de variables "data") se almacenan en las variables "data.x" y "data.ref". En el caso de que sea necesario realizar referenciados para efectuar medidas de zoom, la función, además de estas variables en las que se almacena la potencia espectral para todo el rango de longitudes de onda en el que se ha realizado, también almacenará las potencias espectrales para los tramos de dicho rango definidos por el uso de la función "zz_div.m" (ver sección *5.3.2.6. Generación de tramos para el zoom en las medidas* y *Fragmento de código 16*).

Fragmento de código [16]. Referenciados con opción de zoom habilitada.

```
cd (data.primary_path);
zz_div(center,span,data.zoom_level,0);
cd (data.path);
for i=drange(1:1:data.zoom_level+1)
    switch data.osa
        case 'AGILENT'
            param(1)=data.(sprintf('last_rbw%d',x));
            param(2)=data.(sprintf('last_ref%d',x));
            param(5)=data.(sprintf('last_scale%d',x));
            param(6)=data.(sprintf('last_sens%d',x));
            param(3)=data.(sprintf('zz%d',i))(1);
            param(4)=data.(sprintf('zz%d',i))(2);
            setup(obj,param);
            data.(sprintf('x_ref%d',i))=(linspace(param(3)-param(4)/2,…
            param(3)+param(4)/2,data.N))';
            data.(sprintf('y_ref%d',i))=osa(obj);
        case 'EXFO'
            param(1)=data.(sprintf('zz%d',i))(1);
            param(2)=data.(sprintf('zz%d',i))(2);
            param(3)=data.sens;
            setup(obj,param);
            [data.(sprintf('x_ref%d',i)),...
            data.(sprintf('y_ref%d',i))] = osa(obj);
```

```
            otherwise
                return;
        end
    end
    data.ref=data.(sprintf('y_ref%d',data.zoom_level+1));
    data.x=data.(sprintf('x_ref%d',data.zoom_level+1));
```

Como puede apreciarse en el fragmento de código anterior, debido a que la configuración del analizador es distinta para cada modelo, es necesario hacer uso de una estructura de tipo *switch*, cargando en cada caso, los parámetros necesarios. Antes de realizar los referenciados, se hace uso de la función "zz_div.m" que devuelve en las variables con el prefijo "zz" (*data.zz*) los dos parámetros necesarios para definir los rangos de longitud de onda de los distintos tramos del rango principal en los que se desea medir a modo de zoom. Por otro lado, se observa cómo el código hace uso de los alias *osa* y *setup* que se han generado previamente y que realizan una llamada a las funciones de medida y de configuración respectivamente del OSA con el que se esté trabajando.

Con la salida de la función "referenciador.m" es posible ya proceder a realizar medidas y utilizar las referencias obtenidas para compararlas contra estas medidas.

5.3.2.6 Generación de tramos para el zoom en las medidas.

La generación de tramos dentro del rango de medida para realizar medidas más precisas se realiza mediante la función definida en el archivo "zz_div.m". En la sección anterior, y a lo largo del documento, se han hecho referencias a esta función mediante el término *función de zoom*, que se seguirá empleando a partir de ahora.

La función de zoom, como ya se ha explicado no es sino un divisor de tramos. Esta división de los tramos se realiza a partir del cálculo de nuevos centros y rangos a partir del centro y ancho de rango principal tal y como puede apreciarse en el *Fragmento de código 17*. Ésta función, hace uso de la constante "data.zoom_level" cuyo acceso y manipulación sólo es posible mediante la modificación del código. Ésta variable indica el número de tramos en los que se dividirá el rango introducido.

La función acepta cuatro argumentos de entrada: los parámetros “center” y “span” que definen el rango de medida que se desea dividir en tramos, “zoom_level”, que, como ya se ha especificado, define el número de tramos, y el parámetro “rm” que se utiliza para indicar a la función que debe eliminar de la estructura de variables global los campos generados durante su utilización. Ésta característica se hace necesaria en el caso de que haya varias medidas programadas y se precise llamar a la función “zz_div.m” cada vez que el usuario decide pasar a la siguiente medida. Para ello, cuando el OSA ha terminado de realizar tanto las referencias como las medidas con zoom, se realiza una llamada en la que el parámetro “rm” toma el valor “1” y se suprimen los campos del tipo “data.zz” dejándolos disponibles para la siguiente medida.

La salida de una llamada con el parámetro “rm” tomando el valor “0”, contiene los parámetros “center” y “span” que definen los tramos en los que se debe medir para realizar medidas con zoom, dentro del rango de la medida programada y los almacena en campos de la estructura de variables “data” con el prefijo “zz”.

De forma aclaratoria, se propone como ejemplo un uso de esta función con un valor de “data.zoom_level = 2” y “rm=0”. Con dichos valores de entrada, independientemente de los valores que tomen los parámetros “center” y “span” introducidos en la llamada a la función, se generarán tres campos de tipo “zz”. Los campos “data.zz1” y “data.zz2” contendrán los valores de “center” y “span” que definen los dos tramos en los que se ha dividido el rango de medida y el campo “data.zz3” definirá la medida completa. En el esquema de la *Figura 17*, se representa de forma gráfica la división propuesta como ejemplo. En dicha figura, “Span” y “Centro” definen el rango original mientras que “Span 1” y “Centro 1” por un lado y “Span 2” y “Centro 2” por otro definen los dos nuevos tramos en los que se desea medir.

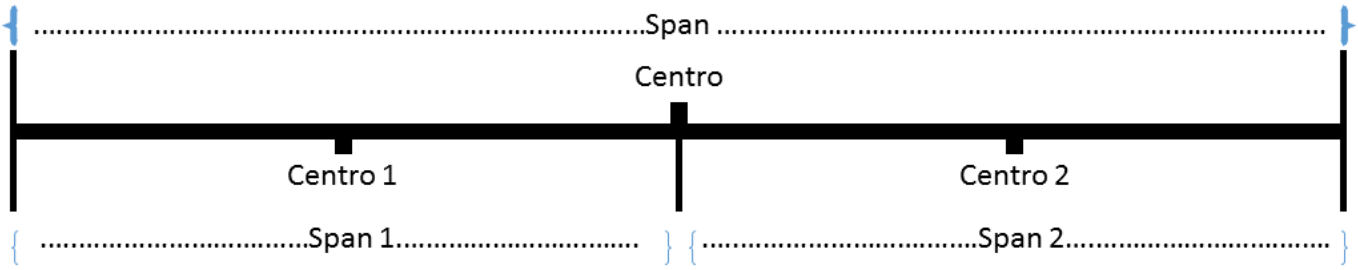

Figura 17. Esquema para el ejemplo de división del rango de medida en tramos.

Fragmento de código [17]. Función "zz_div.m".

```
function zz_div(center, span, zoom_level, rm)
global data
if rm==1
    for x=drange(1:1:zoom_level+1)
        data = rmfield(data, (sprintf('zz%d',x)));
    end
else
    init=center-span/2;
    b=mod(zoom_level,2);
    span2=floor(span/zoom_level);
    for x=drange(1:1:zoom_level+1)
        if x==1
            cent2=init+round(span2/2);
        else
            if b==1
                cent2=cent2+span2+1;
            else
                cent2=cent2+span2;
            end
        end
        cent_toinstr=cent2;
        span_toinstr=span2;
        if x==zoom_level+1
            cent_toinstr=center;
            span_toinstr=span;
        end
        data.(sprintf('zz%d',x))=[cent_toinstr span_toinstr];
    end
end
```

5.3.2.7 Funciones de control de medida.

Las funciones de control de medida, son aquellas que se llaman cuando el usuario interactúa con el diálogo de control de medida. Se han definido tres funciones, una para cada uno de los botones de dicho cuadro de diálogo. Estas funciones únicamente modifican las variables de control de medida para manejar el flujo del programa y pausar, parar o realizar la siguiente medida

programada ante una petición del usuario tal y como se puede observar en los *Fragmentos de código 18, 19 y 20*.

La función "pause_measure.m", únicamente hace uso de la función "dial.m", en la que la instrucción *uiwait* interrumpirá la ejecución de código hasta que el usuario pulse el botón "ok" tal y como ya se ha desarrollado en la sección *5.3.2.3. Función de diálogo*.

La función encargada de redirigir el flujo del programa hacia la siguiente medida programada, se encuentra definida en el archivo "next_measure.m". Las instrucciones de esta función tienen la tarea de comprobar, en primer lugar, si hay programada alguna medida más o si la que se está realizando es la última. Para ello se evalúan las variables "data.cont", que almacena el número indicativo de la medida que se debe realizar y la variable "data.last" que almacena el número de medidas programadas. En el caso de que los valores coincidan, que implica que la medida que se está realizando es la última medida programada, se hace una llamada a la función de interrupción de medidas. En caso contrario se incrementa en uno la variable indicadora de la medida "data.cont" y se asigna a la variable "data.loop" de control del bucle de medidas (puede verse su utilización en el *Fragmento de código 12* dentro de la sección *5.3.2.2. Ventana principal de la interfaz*) el valor "0", para que en la próxima iteración se realice una configuración del analizador antes de solicitar los datos de medida.

Por último la función "stop_loop", asigna a la variable de control de medidas "data.control" el valor "0" consiguiendo que en la siguiente evaluación de esta variable por parte del bucle *while* se produzca una interrupción del mismo.

Fragmento de código [18]. Función "pause_measure.m".

```
function pause_measure()
msg='Pulse ok para continuar';
dial(msg);
```

Fragmento de código [19]. Función "next_measure.m".

```
function next_measure()
global data
if data.cont==data.last
```

```
    delete(gcf);
    stop_loop;
else
    data.cont=data.cont+1;
    data.loop=0;
end
```

Fragmento de código [20]. Función "stop_loop.m".

```
function stop_loop()
global data
data.control=0;
```

El uso de tres archivos distintos para estas funciones responde a la intencionalidad general del software desarrollado y tiene el doble objetivo de ser ampliable por una parte y fácilmente depurable por otra.

5.3.2.8 Función de guardado de datos.

Es la función que se encarga de almacenar todas las medidas realizadas mediante el software. Se encuentra definida en el archivo "guardar.m" y acepta como parámetros de entrada "param", "r", "zoom" e "init". La variable "param" es un vector que contiene los parámetros especificados por el usuario y que definen la configuración utilizada por el analizador que se ha empleado en medir. El tamaño de este vector será variable en función de qué analizador se esté utilizando. Las variables "r", "zoom" e "init" controlan el modo de operación de la función. Como se puede apreciar en el *Fragmento de código 21*, el valor que toma "r" cuando se hace una llamada a esta función determina si es necesario almacenar los parámetros de medida en un archivo distinto. El archivo que contiene los parámetros se crea únicamente cada vez que el software realiza una medida con una configuración distinta y el parámetro "r" garantiza esta posibilidad. El uso del parámetro "zoom" puede verse en el *Fragmento de código 22* y se utiliza para guardar en archivos separados las medidas realizadas cuando el usuario solicita la división del rango de medida en tramos, operación realizada por la función "zz_div.m" (ver sección *5.3.2.6. Generación de tramos para el zoom en las medidas*). Por otro lado, el parámetro de entrada "init" permite a la función de guardado de datos

establecer el identificador que llevará el nombre del archivo. Este identificador no es sino un contador almacenado en la variable "data.id" que indica el número de medidas distintas almacenadas en el directorio de guardado. De esta forma si en la llamada a la función "init" toma el valor "1", se actualizará el valor de dicha variable incrementando en una unidad el valor del identificador de archivo más alto encontrado en el directorio y asignando el nombre al archivo de la forma "idID_OSA.mat", donde "ID" es el identificador de medida y "OSA" es el modelo de analizador utilizado en dicha medida. La llamada a "guardar.m" con "init" fijado en "1", es decir, la creación de un archivo de guardado distinto, se realiza cada vez que el usuario pulsa el botón "medir" y por ello únicamente en la primera iteración del bucle de medidas. El resto de las medidas realizadas hasta que se solicite una parada únicamente actualizarán los datos del archivo.

La llamada a esta función se realiza desde la función de gestión de medidas de cada analizador cada vez que el OSA ha efectuado una medida de forma satisfactoria con el objetivo de conservar todas las medidas realizadas. El formato de los archivos de salida es tanto ".mat" como ".txt". Los archivos que contienen los datos obtenidos en la medición, se almacenan en formato ".mat" y el formato ".txt" se utiliza para almacenar los parámetros de configuración que se han utilizado en la medición. Los parámetros se almacenan en formato ".txt" para permitir al usuario un acceso a estos archivos desde cualquier lugar, dada la universalidad de la extensión de texto plano. Por otro lado el formato ".mat" es la extensión utilizada por Matlab por defecto y permite la carga directa de las variables almacenadas en el espacio de trabajo desde el navegador de archivos que proporciona el entorno de desarrollo y por ello se ha elegido dicho formato para almacenar los vectores de datos.

Fragmento de código [21]. Guardado de parámetros en un archivo separado.

```
if r==1
    filename=sprintf('id%d_%s_parameters.txt',data.id,data.osa);
    file=[PathFolder '/' filename];
    str=zeros(length(param),1);
    for i=drange(1:1:length(param))
        str(i,1)=param(i);
    end
    save(file, 'str', '-ASCII','-append');
end
```

Fragmento de código [22]. Guardado de medidas realizadas con el modo "zoom" activado.

```
if zoom == 1
    filename = sprintf('id%d_zoom_%s_%s.mat', data.id,time_str,data.osa);
    A = 0;
    for i=drange(1:1:data.zoom_level)
        file = [PathFolder '/' filename];
        x_z=data.(sprintf('x_med%d',i));
        y_z=data.(sprintf('y_save%d',i));
        s = size(A);
        if s(1) == s(2)
            A(:,1)=x_z;
            A(:,2)=y_z;
        else
            A(:,s+1)=x_z;
            A(:,s+2)=y_z;
        end
    end
    save(file, 'A');
```

5.3.2.9 Exportación de datos.

Las últimas dos funciones del bloque común, que se encargan de exportar los datos que se le indiquen en el formato de Microsoft Office ".xlsx" son "excel_export.m" y "excel_constructor.m". El código que se encuentra en "excel_export.m" maneja la figura generada a partir de la herramienta *guide* en el archivo "excel_export.fig" y hace uso de la función "excel_constructor.m" que prepara los datos y realiza la exportación.

La función se llama desde la ventana principal de la interfaz y responde a la petición de ejecución por parte del *callback* correspondiente al botón "Exportar medidas", tal y cómo se especifica en la sección *5.3.2.2. Ventana principal de la interfaz*.

La ventana que permite al usuario interactuar con el software para exportar los archivos de medida generados, consta de un *pop-up* de selección y de los dos botones de control “Aceptar” y “Salir” tal y como se observa en la *Figura 18*.

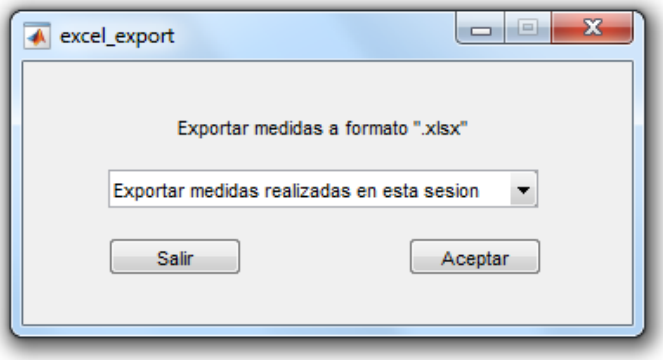

Figura 18. Diálogo de exportación de medidas

En el *pop-up* de selección el usuario puede elegir entre realizar una exportación de todas las medidas realizadas durante la sesión actual o elegir un directorio de archivos de medida personalizado. Cuando se ha efectuado la selección y se pulsa el botón de aceptar, se ejecuta la función de *callback* correspondiente (ver *Fragmento de código 23*).

Fragmento de código [23]. Callback para el botón aceptar en la ventana de exportación de medidas.

```
function button_aceptar_Callback(hObject, eventdata, handles)
choice=get(handles.popup,'value');
DateString = datestr(datetime('today'));
[stat,struc] = fileattrib;
PathCurrent = struc.Name;
if choice==1
    PathFolder = [PathCurrent '\medidas_realizadas\' DateString];
else if choice==2
        PathFolder=uigetdir(PathCurrent,'Seleccione el directorio con las
medidas que desea exportar');
    end
end
[PathFile]=uigetdir(PathCurrent,'Seleccione donde desea guardar las medidas
exportadas');
```

```
Path=[PathFile '\RESULTADOS' '\' DateString];
if exist([PathFile '\RESULTADOS' '\' DateString],'file')==0
    mkdir(Path);
end
try
    excel_constructor(PathFolder,Path);
    delete(excel_export);
    msg='Se ha terminado de exportar..';
    dial(msg);
catch
    msg='ERROR: No se ha podido exportar..';
    dial(msg);
end
```

En primer lugar, la función evalúa la opción seleccionada y asigna un valor a la variable "PathFolder" que contiene la ruta del directorio con los ficheros de medidas que se desean exportar, ya sea el directorio de guardado automático de medidas o el que el usuario haya elegido. Acto seguido se asigna a la variable "PathFile" la ruta en la que el usuario desee guardar los archivos exportados. Se genera de forma automática un directorio con el nombre "RESULTADOS" y un sub-directorio con la fecha de la exportación donde finalmente de crearán los archivos con extensión ".xlsx".

La exportación de archivos se realiza mediante la función "excel_constructor" que, como puede observarse en la llamada que se efectúa en el *Fragmento de código 23*, precisa de dos argumentos de entrada, la ruta del directorio con los archivos de medida y la ruta del directorio de salida de archivos.

La primera parte de la función de exportación busca las distintas "ids" (ver sección *5.3.2.1. Inicialización del software*) que corresponden a archivos de medida distintos dentro del directorio y las almacena en la variable "id". Como puede apreciarse en el *Fragmento de código 24*, la búsqueda de la cadena de caracteres ".mat" en el nombre de cada archivo en el directorio de entrada, garantiza que se obtienen todas las "ids" distintas que pueda haber, ya que el archivo de almacenado de parámetros como se explica en la sección *5.3.2.8. Función de guardado*, se almacena en formato ".txt". De esta forma, en el bucle de tipo *for* la variable "n" indica el número de archivos del directorio y la variable "a" contiene la estructura de archivos de dicho directorio.

Si se analiza detenidamente el código y la variable "id" de salida del bucle utilizado, se observa que se ha optado por no pre-almacenar espacio de memoria, a pesar de que se van añadiendo valores en cada búsqueda satisfactoria. El pre-almacenamiento de memoria para rutinas iterativas que actualizan valores de una variable siempre es recomendado ya que incrementa la velocidad de ejecución de forma importante, en especial si se trata de gran cantidad de información. En este caso, se considera innecesario ya que para poder prealmacenar espacio para dicha variable sería necesario realizar otro bucle de tipo *for* y una nueva variable que contase el número de archivos de medida reales que hay en el directorio, haciendo del prealmacenamiento de memoria una tarea inútil.

Fragmento de código [24]. Búsqueda de distintas "ids" en el directorio.

```
a=dir(path_mat);
n=length(a);
id='';
id=cellstr(id);
for i = drange(1:1:n)
    if strfind(a(i).name,'.mat')~=0
        p_ = strfind(a(i).name,'_');
        id = [id; a(i).name(3:p_(1)-1)];
    end
end
```

La segunda parte del código, va buscando cada una de las "ids" encontradas en el bucle anterior en los nombres de todos los archivos del directorio. En primer lugar, obtiene los parámetros de configuración almacenados en el archivo que contiene la cadena de caracteres "_parameters.txt" y, en segundo lugar, los vectores de salida de las medidas (ver *Fragmento de código 25*).

Se generan tantos archivos de salida ".xlsx" como "ids" distintas se hayan encontrado y cada uno de estos archivos contiene tantas columnas como medidas se hayan realizado con la misma id, además de la columna con el vector de longitudes de onda. Así mismo, cada libro se construye con un encabezado que contiene el nombre del modelo de analizador utilizado y los parámetros

utilizados en la medida. El nombre de los archivos que se generan es el mismo que el nombre de los archivos de partida.

Se ha programado así mismo una barra de progreso que informa de forma gráfica al usuario del progreso de la exportación de los archivos (*Figura 19*).

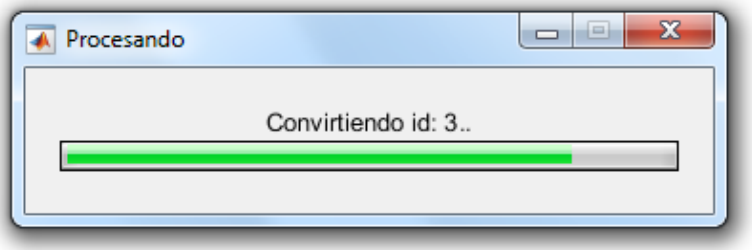

Figura 19. Barra de progreso de la exportación.

Fragmento de código [25]. Preparación de archivo y exportación iterativa a formato ".xlsx".

```
PARAM=struct;
for k = drange(2:1:length(id))
    id_str=id{k};
    for i=drange(1:1:n)
        name = a(i).name;
        if strfind(name,id_str) ~= 0
            p_param = strfind(name,'_parameters.txt');
            if  p_param~=0
                pos_ = strfind(name,'_');
                osa = name(pos_(1)+1:p_param-1);
                B = load([path_mat '/' name]); l = length(B);
                PARAM.(sprintf('file_name%d',k)) = sprintf('%s_%s.xlsx',
                                                   id_str,osa);
                head(1,1)={'Model'};head(1,2)={osa};head(2,1)={'Parameters'};
                head(4,1)={'Wavelength (nm)'};head(4,2)={'Measures (dB)...'};
                for x = drange(2:1:l+1); head(2,x)={B(x-1)}; end
                PARAM.(sprintf('head%d',k))=head;
                PARAM.(sprintf('name%d',k))=name;
            end
        end
    end
```

```
end
h = waitbar(0,sprintf('Convirtiendo id: %s..',id_str),'Name','Procesando');
for k = drange(2:1:length(id))
    id_str=id{k};
    for i=drange(1:1:n)
        waitbar(i/n); name = a(i).name;
        if strfind(name,id_str) ~= 0
            if strfind(name,'.mat')~=0
                xlswrite([path_xlsx '/' PARAM.(sprintf('file_name%d',k))],
                         PARAM.(sprintf('head%d',k)),1,'A1');
                Bdata=load([path_mat '/' name]); Bdata = Bdata.A;
                xlswrite([path_xlsx '/' PARAM.(sprintf('file_name%d',k))],
                         Bdata,1,'A4');
            end
        end
    end
end
close (h);
```

Con esta función, se termina la descripción del bloque común del software, es decir, de todas aquellas funciones que son parte fundamental de su funcionamiento y que deben ser ejecutadas independientemente de qué modelo de analizador se esté utilizando.

5.3.3 Funciones propias de cada analizador.

Esta sección del documento, se encargará de describir qué funciones son necesarias pero independientes para cada analizador para el que se ha configurado la interfaz gráfica de usuario desarrollada.

5.3.3.1 Función de inicialización del OSA.

La primera función indispensable para el control de un OSA es la función que se encarga de inicializar correctamente el instrumento, es decir, de establecer una comunicación satisfactoria y prepararlo para realizar mediciones. Esta función es la que conecta el software con el hardware utilizado para la comunicación, en este caso el bus GPIB. Tal y como se ha estructurado el software,

el nombre del archivo que define esta función tiene el formato “modelo_init.m”, donde la cadena “modelo” es el nombre que identifica el analizador.

En la *Figura 20* vemos las rutinas que debe contener esta función para una correcta integración dentro de la interfaz.

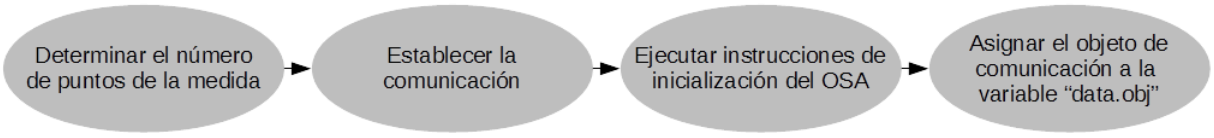

Figura 20. Diagrama de flujo de la función de inicialización del OSA.

La primera acción que debe realizarse es la determinación del número de puntos que va a suministrar el instrumento, almacenando el valor en la variable “data.N”. La modificación de este valor únicamente puede realizarse mediante la modificación del código de esta función y se ha establecido por defecto en el valor máximo admitido por cada analizador.

Una vez identificados, instalados y configurados los drivers de dispositivo necesarios para detectar la tarjeta PCI-GPIB o la interfaz utilizada para conectar el dispositivo de medida con el de control, la primera condición que debe establecerse es la correcta comunicación con el analizador y la creación de un objeto de comunicación en Matlab a través del comando *gpib* perteneciente al *Instrument Control Toolbox*. El bloque de establecimiento de la comunicación puede observarse en el *Fragmento de código 26* que corresponde a las instrucciones utilizadas en el caso del OSA EXFO. La primera instrucción corresponde a la creación de un objeto *gpib*. Para ello, es necesario especificar tres parámetros. El primero de ellos, “vendor”. Se trata de una denominación asignada en la documentación de Matlab. Es un identificador del fabricante del hardware que se encarga de la comunicación, en este caso 'ni' (que corresponde a *National Instruments*), fabricante de la tarjeta PCI-GPIB del equipo controlador. El segundo, “boardindex”, especifica el primer parámetro de la comunicación GPIB. Por último “primaryaddress” indica qué dirección GPIB tiene asignada el instrumento [9].

Una vez que se ha creado el objeto de comunicación, antes de utilizar el comando *fopen*, para conectar dicho objeto al instrumento, hay que asignar un espacio de memoria para los datos de entrada cuando se realice cualquier comunicación con el instrumento (la operación que hará un mayor uso de este *buffer* será la de adquisición de los datos de medida) y para ello se añade un campo al objeto creado con el nombre '*InputBufferSize'.* El tamaño de este *buffer* se ha mantenido idéntico al tamaño utilizado en el programa heredado.

De esta forma, con estas tres sencillas lineas, se efectúa una comunicación satisfactoria con el instrumento y se está en disposición de comenzar a enviar y recibir información.

Fragmento de código [26]. Instrucciones de inicialización de la comunicación para el OSA EXFO.

```
osa = gpib('ni', 0, 5);
set(osa, 'InputBufferSize',2001*17);
fopen(osa);
```

El tercer bloque de la función debe enviar al instrumento las instrucciones SCPI que realicen la rutina de inicialización del OSA y lo preparen para realizar medidas. Estas instrucciones se envían al instrumento a través del uso del comando *fprintf*, utilizando dos de los parámetros de entrada que acepta. La sintaxis utilizada por ello es: `fprintf(obj,'cmd'),` donde "obj" es el objeto de comunicación *gpib* y "cmd" es la cadena de caracteres con la instrucción SCPI que se desea enviar al analizador [22]. Las instrucciones concretas que envía tanto esta función como la de configuración y medida de los dos modelos de OSA utilizados para el desarrollo de la interfaz se adjuntan en la sección de los anexos *11.1. Instrucciones SCPI para la inicialización, configuración y medida necesarias en los modelos de OSA para los que se ha configurado la interfaz*.

Por último, se almacena en la variable "data.obj" el objeto de comunicación creado.

5.3.3.2 Función de configuración del OSA.

Las funciones de configuración de un OSA que se desee controlar mediante la interfaz desarrollada son nombradas de la forma "modelo_setup.m" donde al igual que en la función de inicialización, "modelo" es el identificador del OSA. (Por ejemplo, en el caso del analizador AGILENT la función se nombra "AGILENT_setup.m".

Esta función debe aceptar como entrada dos variables, el objeto de comunicación "osa" y el vector de parámetros de configuración "param" que contiene los valores de las opciones de medida que el usuario haya seleccionado para la medida que se realizará tras la ejecución correcta de las instrucciones de configuración (ver sección 5.3.3.4. *Ventana de parámetros*). De esta forma, el código que se ejecuta no es sino una serie de instrucciones *fprintf* que envían las instrucciones SCPI con los parámetros contenidos en el vector "param". En el *Fragmento de código 27* puede verse la función de configuración del AGILENT 86142A a modo de ejemplo.

Fragmento de código [27]. Función de configuración del AGILENT 86142A.

```
function AGILENT_setup(osa,param)
fprintf(osa, ['Sens:Wav:Cent ',num2str(param(3)),' nm']);
fprintf(osa, ['Sens:Wav:span ',num2str(param(4)),' nm']);
fprintf(osa, ['Sens:Bwid:Res ',num2str(param(1)),' nm']);
fprintf(osa, ['Disp:Wind:Trac:Y:Scal:Rlev ',num2str(param(2)),' dBm']);
fprintf(osa, ['DISP:WIND:TRAC:Y:SCAL:PDIV ',num2str(param(5))]);
fprintf(osa, ['Sens:Pow:Dc:Rang:Low ',num2str(param(6)),' dBm']);
```

5.3.3.3 Función de medida.

La función de medida es la función que debe enviar al instrumento las instrucciones SCPI para la solicitud de datos, leer dichos datos y generar una salida con el vector de potencia espectral.

Para poder ejecutar una orden de medida en el analizador, es necesario haber realizado anteriormente una llamada a la función de configuración del OSA. Dependiendo de las instrucciones soportadas por el instrumento que se desea controlar, la estructura de la función de medida variará sensiblemente.

Tomando como referencia los analizadores utilizados para el desarrollo de la interfaz, en el caso del analizador AGILENT 86142A la función se reduce a tres líneas. Como puede observarse en el *Fragmento de código 28*, se realiza una petición de datos mediante la instrucción *query* y se almacenan en la variable "yhOSA", se sustituye el carácter separador "," por un espacio en blanco y se genera el vector de salida "espectro" con los valores numéricos en formato "float".

Sin embargo, para el modelo con el que se realizaron las primeras pruebas, el analizador EXFO, la función de medida presenta un diagrama de flujo más complejo que se explicará a continuación. Este modelo, únicamente puede proporcionar los datos obtenidos en la medida en bloques de 2000 puntos y, por ello, aunque la instrucción SCPI mediante la cual se le deben solicitar dichos datos es la misma, el parámetro indicador de los puntos que se solicitan varía y se hace necesario un proceso iterativo. Por otro lado, para que el analizador realice las medidas, es necesario enviarle una petición de inicio de adquisición de datos mediante la instrucción SCPI "INIT0:AUTO 1" tal y como puede observarse en el *Fragmento de código 29*. Una vez el analizador se encuentra preparado para suministrar los datos, se ejecuta una rutina iterativa mediante el uso de un bucle de tipo *for* que realizará las iteraciones en función del número de puntos que se deseen obtener.

Fragmento de código [28]. Función de medida para el AGILENT 86142A.

```
function espectro = AGILENT_measure(osa)
yhOSA  = query(osa, 'Trac:Data:Y? TrA');
yhOSA =  strrep(yhOSA, ',', ' ');
espectro = sscanf(yhOSA,'%f');
```

Fragmento de código [29]. Función de medida para el EXFO.

```
function espectro = EXFO_measure(EXFO)
global data;
alm=0;
fprintf(EXFO, 'INIT0:AUTO 1\n');
fprintf(EXFO, 'INIT0:ACQ:STAT?\n');
alm = str2double(fgetl(EXFO));
pausa=0.01;
```

```
while alm==0
    fprintf(EXFO, 'INIT0:ACQ:STAT?\n');
    alm = str2double(fgetl(EXFO));
    pause(pausa);
end

pnt=(data.N-2000)/1000;

for i=drange(0:1:pnt/2)
    pause(pausa);
    n=num2str(i*1000*2); com = sprintf('TRAC0:DATA? %s\n',n);
    fprintf(EXFO, '%s', com); pause(pausa);
    alm = fgetl(EXFO); alm =  strrep(alm, ',', ' ');
    if i==0
        y = sscanf(alm,'%f');
    else
        y = [y;sscanf(alm,'%f')];
    end
end
espectro=y;
```

5.3.3.4 Ventana de parámetros.

La ventana de parámetros contiene todos los elementos gráficos necesarios para permitir al usuario configurar las medidas que desee realizar. La ventana se encuentra definida en la figura "set_parameters.fig" generada mediante la herramienta *guide* y es controlada por la función contenida en el archivo "set_parameters.m".

Las opciones que ofrece la ventana de configuración de medida varían dependiendo de las posibilidades de configuración que se desean implementar a cada analizador. Sin embargo, el funcionamiento esencial puede representarse fácilmente como se observa en la *Figura 21*.

La salida que genera la función de *callback* del botón de aceptar de la ventana de configuración debe ser un conjunto de variables que contengan todos los parámetros necesarios para la configuración del analizador que se esté utilizando. En la sección *6. Resultados y demostración del funcionamiento de la interfaz* se documenta un ejemplo de utilización de la

interfaz y en ella se puede observar la forma de operar de esta ventana.

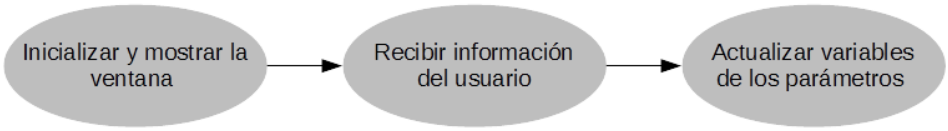

Figura 21. Diagrama de flujo simplificado de la función "set_parameters.m".

Las opciones mínimas de configuración que deberá ofrecer esta ventana de configuración para cualquier OSA que se desee controlar son los parámetros “center” y “span”, que definen el rango en el que se desea efectuar la medida. A continuación, se procederá a detallar las opciones de configuración implementadas en las ventanas de configuración de medidas de los dos analizadores utilizados para el desarrollo de la interfaz gráfica de usuario diseñada.

En el caso del analizador EXFO, las opciones de configuración que se han implementado además de la definición del rango de longitudes de onda en las que se desea medir son: la elección de la sensibilidad del OSA en un rango del 1 al 9 (configuración que también se efectúa en el programa heredado) y la posibilidad de realizar un zoom en la medida programada.

La ventana del modelo AGILENT 86142A permite configurar, además del rango de medida, la resolución de ancho de banda, el nivel de ganancia de la referencia, la escala de medida, el nivel de sensibilidad y, al igual que en el EXFO, la realización de zoom en las medidas.

5.3.3.5 Función principal de gestión del analizador.

Una vez descrito el código necesario para la correcta inicialización, configuración y toma de medidas mediante un analizador de espectros, se hace necesario desarrollar una función que se encargue de automatizar las llamadas a dichas funciones de operación y que permita una interacción lo más sencilla posible con la ventana principal de la interfaz. Esta función, por ello, cumple el cometido de gestionar las operaciones que involucran directamente al analizador, tanto en su configuración como en la solicitud de datos, y es la función a la que se llama desde el bucle de medidas de la función de *callback* del botón de medir de la ventana principal.

La función se nombra "modelo_start.m" siendo "modelo" la identificación del analizador que se esté utilizando. Como se puede observar en el *Fragmento de código 12* dentro de la sección *5.3.2.2. Ventana principal de la interfaz,* admite el parámetro "loop", que indica a la función la necesidad o no de realizar una configuración del OSA tal y como se detalla en la descripción del bucle de medidas incluido así mismo en dicha sección.

Se justifica el desarrollo de una función de gestión por cada analizador debido a tres razones fundamentales:

1. Las distintas necesidades de configuración de cada analizador.
2. Necesidad de un distinto procesado de los datos obtenidos mediante la función de medida.
3. Favorece la posibilidad de ampliación y la característica de universalidad del software desarrollado.

El cometido de la función es generar, teniendo en cuenta todas las condiciones fijadas tanto por las variables de flujo del software como por el usuario, los dos vectores de información que se desean visualizar en la ventana principal.

El diagrama de flujo puede verse en la *Figura 22*. Además, en el *Fragmento de código 30*, puede consultarse la implementación práctica de dicho diagrama, tomando como ejemplo la función de gestión desarrollada para el analizador AGILENT 86142A.

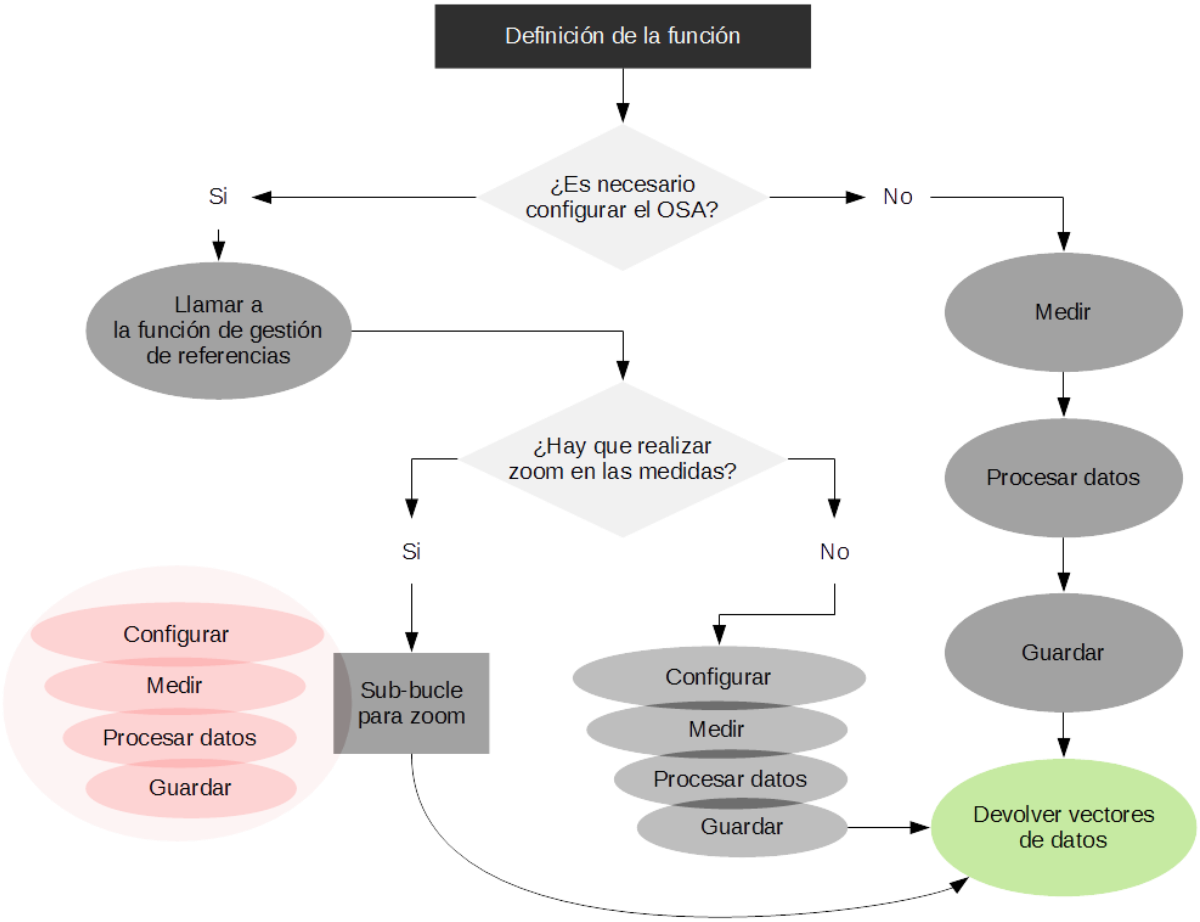

Figura 22. Diagrama de flujo de la función de gestión.

Como se observa en el *Fragmento de código 30*, los vectores de salida de la función son los valores que el bucle de medidas de la función principal procederá a representar en los ejes de coordenadas de la ventana principal.

Fragmento de código [30]. Ejemplo de la función de gestión para el analizador AGILENT 86142A.

```
function [x, y_plot] = AGILENT_start(loop)
global data
hOSA=data.obj;
i=data.cont;
if loop==0
    cd (data.primary_path);
    try
```

```
        referenciador(hOSA,data.zoom,data.(sprintf('last_center%d',i)),...
        data.(sprintf('last_span%d',i)));
    catch
        data.error=[data.error 'Referencias no cargadas correctamente'];
        return;
    end
    try
        msg='Pulse ok para continuar con la toma de espectros';
        cd (data.primary_path);
        dial(msg);
        cd (data.path);
        if data.zoom==1
            for x=drange(1:1:data.zoom_level+1)
                param(1)=data.(sprintf('last_rbw%d',i));
                param(2)=data.(sprintf('last_ref%d',i));
                param(5)=data.(sprintf('last_scale%d',i));
                param(6)=data.(sprintf('last_sens%d',i));
                param(3)=data.(sprintf('zz%d',x))(1);
                param(4)=data.(sprintf('zz%d',x))(2);
                AGILENT_setup(hOSA,param);
                data.(sprintf('x_med%d',x))=linspace(param(3)-param(4)/2,...
                param(3)+param(4)/2,data.N)';
                data.(sprintf('y_med%d',x))=AGILENT_measure(hOSA);
                data.(sprintf('y_save%d',x))=smooth(data.(sprintf('y_med%d',x))-
data.(sprintf('y_ref%d',x)),data.smooth);
            end
            cd (data.primary_path);
            zz_div(data.(sprintf('last_center%d',i)),...
            data.(sprintf('last_span%d',i)),data.zoom_level,1);
            cd (data.path);
            y=data.(sprintf('y_med%d',zoom_level+1));
            cd (data.primary_path);
            guardar(data.x,data.(sprintf('y_save%d',zoom_level+1)),param,1,1);
            cd (data.path);
        else
            param(1)=data.(sprintf('last_rbw%d',i));
            param(2)=data.(sprintf('last_ref%d',i));
            param(3)=data.(sprintf('last_center%d',i));
            param(4)=data.(sprintf('last_span%d',i));
            param(5)=data.(sprintf('last_scale%d',i));
            param(6)=data.(sprintf('last_sens%d',i));
```

```
            AGILENT_setup(hOSA,param);
            y=AGILENT_measure(hOSA);
            cd (data.primary_path);
            guardar(data.x,smooth(y-data.ref,data.smooth),param,1,0);
            cd (data.path);
        end
    catch
        data.error=[data.error 'No se ha podido medir'];
        return;
    end
else
    y=AGILENT_measure(hOSA);
    cd (data.primary_path);
    guardar(data.x,smooth(y-data.ref,data.smooth),0,0,0);
    cd (data.path);
end
x=data.x;
y_plot=smooth(y-data.ref,data.smooth);
```

5.3.3.6 Procesado de los datos.

El procesado de los datos suministrados por la función de medida depende de cada OSA, y, como ya se ha discutido en la sección *5.3.3.5. Función principal de gestión del analizador*, es una de las razones por las que no se ha realizado una unificación de dicha función.

Este tratamiento de los datos tiene como objetivo realizar la comparación de la medida obtenida con la medida tomada como referencia, así como efectuar un suavizado de la curva obtenida para facilitar su visualización. El suavizado de los datos se realiza mediante el uso de la instrucción *smooth* dentro del *Curve Fitting Toolbox* [23]. Esta función, tal y como se puede extraer de la documentación de Matlab, puede utilizar varios métodos de suavizado. En este caso se ha optado por utilizar una media móvil, a modo de filtro paso bajo, requiriéndose el parámetro *span* para especificar el nivel de suavizado. Este parámetro lo fija el usuario en la primera ventana de configuración de la interfaz tal y como se explica en la sección *5.3.2.1. Inicialización del Software* dentro del bloque *Funciones comunes*.

Como alternativa a la utilización de esta función, se desarrolló una función que realiza el promediado de los valores con un parámetro de "span" interpolando entre los valores separados por dicho "span". Tras realizar una comparativa entre los tiempos de ejecución de ambas funciones se decidió utilizar la función *smooth* por ser, de media, en torno a 1 décima de segundo más rápida (tiempo de ejecución función *smooth*: 0.048696 s, tiempo de ejecución función de promediado desarrollada: 0.148142 s).

5.3.4 Gestión de errores y excepciones durante la ejecución del software.

Los errores o excepciones de ejecución que se produzcan durante el desarrollo del software se gestionan mediante el uso de dos herramientas. Tras terminar el proceso de depurado de código, se ha encapsulado el código en estructuras de tipo *try/catch*. La segunda herramienta de gestión, ha sido el uso de la variable "data.error" para almacenar en forma de cadenas de caracteres un indicador del error, de tal forma que cuando se resuelve la última de las estructuras *try/catch* se muestra al usuario un mensaje a través del uso de la función "dial.m" con el indicador del error que se ha producido [24].

6. Resultados y demostración del funcionamiento de la interfaz.

En esta sección se documentará un ejemplo de la utilización del software desarrollado para la obtención de medidas.

6.1 Configuración utilizada.

Se desean obtener las medidas de la potencia espectral frente a las longitudes de onda de la señal que circula por una fibra LPFG que presenta una longitud de onda de resonancia en torno a los 1525nm. Para ello se dispone del analizador AGILENT 86142A y la fuente de luz AGILENT presentes en el laboratorio. La conexión con el equipo de medida se realiza a través del bus GPIB y la una tarjeta GPIB-PCI.

6.2 Procedimiento de medida.

En primer lugar, y tras encender los equipos, se inicia la interfaz desde el IDE de Matlab situando el *workspace* en el directorio principal del programa e iniciando la función “init.m”.

Tras un aviso, que recomienda al usuario leer el archivo “readme.txt”, presente en dicho directorio y en el que se indican las instrucciones básicas de inicialización y manejo de la interfaz (*Figura 23)*, se solicita la elección de OSA y la gestión de referencias.

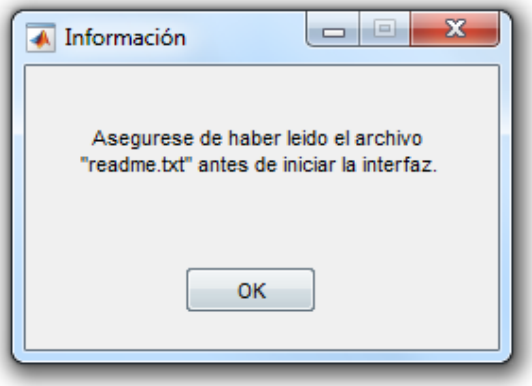

Figura 23. Ventana informativa.

Se escoge el analizador AGILENT 86142A y la opción de realizar un referenciado por cada medida, así como un nivel de suavizado de las curvas de "1" (*Figuras 24 y 25*).

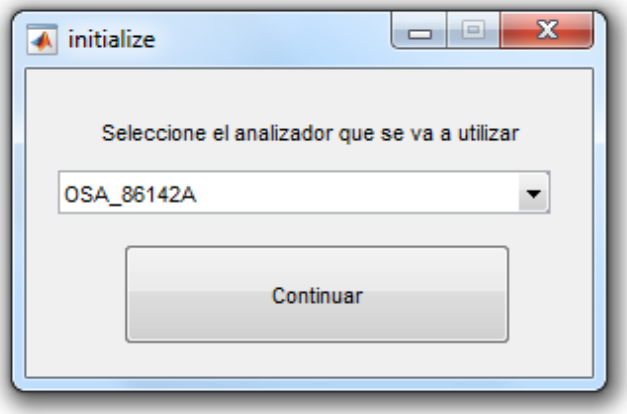

Figura 24. Selección OSA.

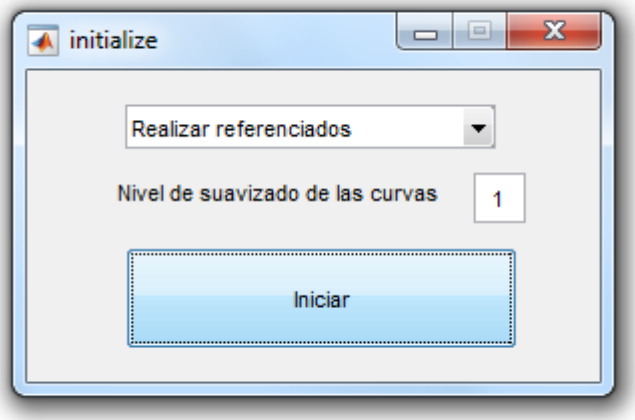

Figura 25. Referencias.

Una vez realizada la configuración inicial, se abre la ventana principal de la interfaz que, como se observa en la *Figura 26*, no permite comenzar la medida, puesto que primero hay que configurarla.

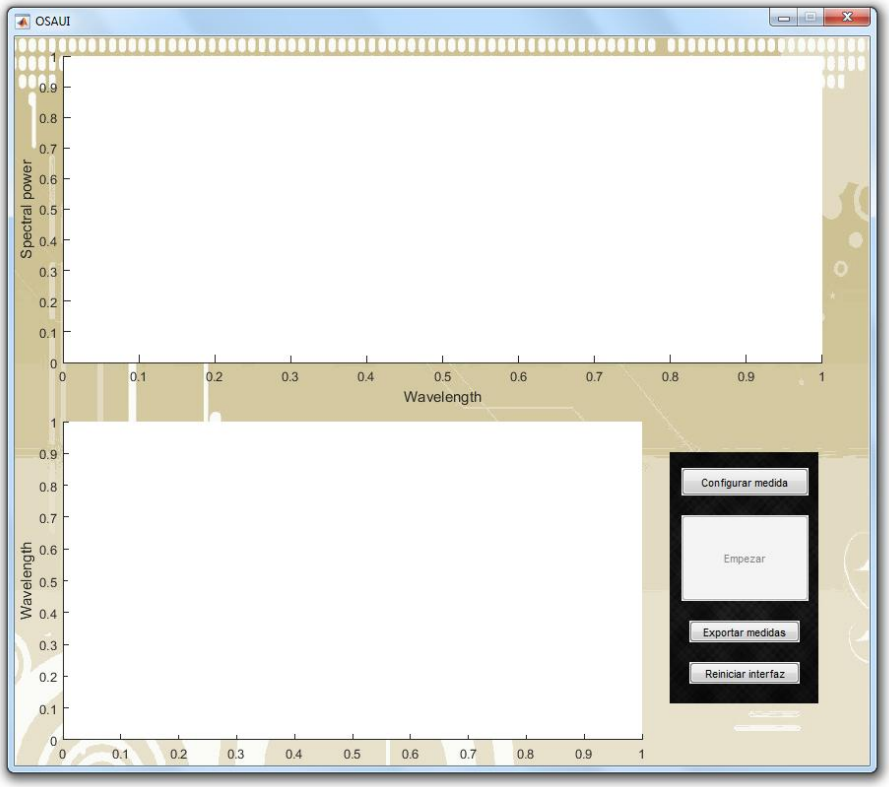

Figura 26. Ventana principal tras la inicialización.

Para ello, se presiona el botón "Configurar medida" con lo que se abre la ventana de configuración. Como se observa en la *Figura 27,* para esta demostración se ha escogido realizar una medida sin zoom y con los siguientes valores para los parámetros de configuración:

RBW = 5; Centro = 1425; Span = 300;

Escala = 5; Sensibilidad = -70;

Nivel de referencia = -28;

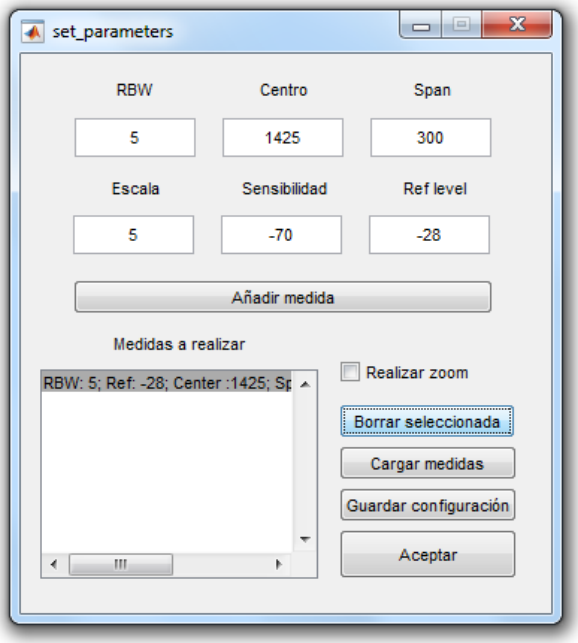

Figura 27. Configuración de la medida.

Tras realizar la configuración de la medida, se pulsa el botón de aceptar. A continuación, se pulsará el botón "Empezar" para que comience el proceso de medición.

Al presionar el botón de comenzar medida, se genera el cuadro de control de medida y aparece un diálogo informando de que el software se encuentra preparado para realizar la captura de referencia.

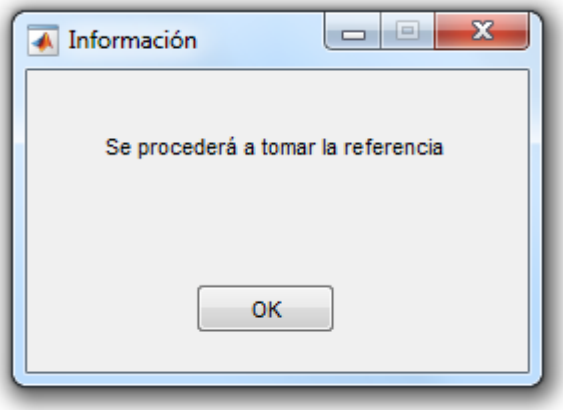

Figura 28. Esperando al usuario para tomar una referencia.

Tras conectar la fuente de luz con la que se va a operar directamente al analizador, se pulsa el botón "ok" y el programa comenzará a obtener los datos de referencia (*Figura 28*).

Cuando se haya obtenido dicha referencia, otro cuadro de diálogo espera a que el usuario pulse el botón de "ok" para comenzar a medir. Cuando la referencia esta capturada, se realiza el conexionado que se desee entre la fuente de luz y el analizador y se indica al software que comience a realizar capturas de datos.

6.3 Resultados.

Se muestra en la *Figura 29* la primera medida realizada en color verde que se mantiene en pantalla durante el tiempo que dure la medición. También se muestra una de las medidas obtenidas en color rojo.

En la *Figura 30,* se observa el mapa de colores elaborado durante la medida de demostración a lo largo de 41 capturas.

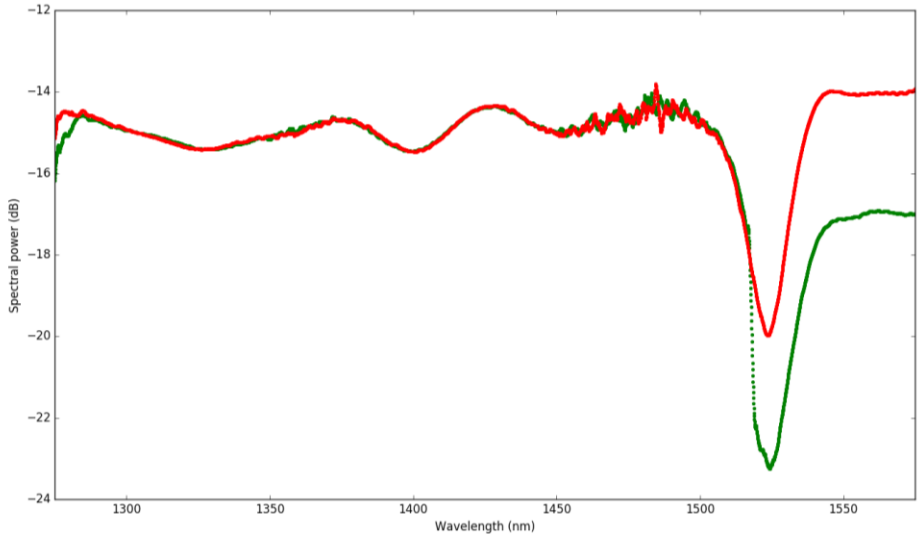

Figura 29. Medida de la potencia espectral.

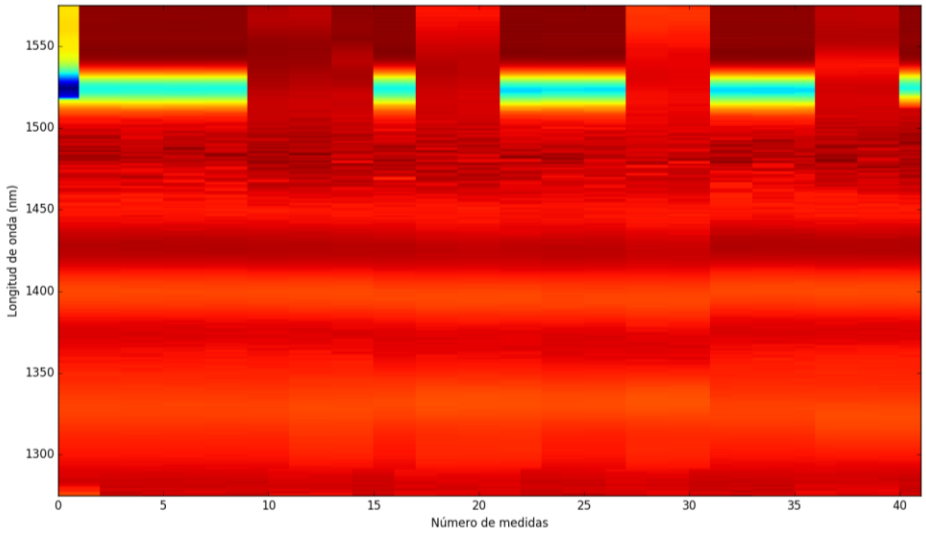

Figura 30. Mapa de colores para la medida demostrativa.

Entre la captura 0 y la 8 la fibra estaba doblada y se veía una resonancia. A continuación, entre las capturas 10 y 15 se tensa la fibra y dicha resonancia desaparece. Esta operación de curvar y tensar se repitió varias veces a lo largo de la obtención de datos.

7. Problemáticas encontradas en el uso de Matlab como entorno de desarrollo

Tras el desarrollo de la interfaz gráfica y su documentación mediante la elaboración de este texto, se ponen de manifiesto las dificultades y problemáticas encontradas en el uso de Matlab como entorno de desarrollo.

1. Comunicación y direcciones GPIB.

   La creación del objeto de comunicación de tipo *gpib* en Matlab, precisa de introducir como parámetro la dirección GPIB asociada al instrumento que se desea controlar. Este parámetro es fijo, es decir que, si esta dirección cambia, ya sea por una actualización o una sustitución del hardware utilizado para la comunicación, es necesario modificar dicho parámetro, mediante la edición del código.

2. Matlab no es un lenguaje de programación de propósito general.

   Con el objetivo de garantizar una buena adaptación de nuevos modelos a la interfaz, así como incrementar su eficiencia, aparece la necesidad de realizar una programación orientada a objectos y para ello modificar la estructura del código.

   El desarrollo de un software basado en la creación de clases en vez de la estructura funcional utilizada supondría dos cambios fundamentales. En primer lugar, para ampliar el abanico de analizadores soportados, bastaría con añadir los métodos correspondientes a una clase general, sin la necesidad de modificar el código principal. En segundo lugar, la estructura de variables global de la que se hace uso podría suprimirse y bastaría con añadir propiedades a los objetos creados.

   Sin embargo, este paradigma de programación no está bien integrado en el lenguaje de Matlab que se basa en la programación funcional. Todo es una función y por ello, aunque en las últimas versiones se haya añadido la posibilidad de crear clases y objetos, su utilización no sería eficiente ya que todos los *Toolbox* que ofrece Mathworks siguen estando basados en funciones.

   Por otro lado, no presenta ningún tipo de optimización automática, el intérprete de código ejecuta literalmente las líneas escritas y Matlab sólo optimiza las funciones si son compiladas. De esta forma, la velocidad de ejecución depende del estilo del programador.

3. Necesidad de múltiples hilos de ejecución.

La implementación de múltiples hilos de ejecución en Matlab es, sencillamente, deficiente. Se presentan dos opciones, la primera de ellas es hacer uso del paquete *Parallel Computing Toolbox* (PCT) y la segunda utilizar la gestión de múltiples hilos de ejecución que ofrece Java. Matlab se sirve de Java para realizar numerosas tareas entre las que se incluye el manejo de GUIs, al igual que ciertas operaciones de procesado de datos. (De hecho, Matlab ejecuta su propio JVM -*Java Virtual Machine* en su inicialización).

El uso del paquete PCT, además de su coste económico, no ofrece sino la posibilidad de ejecutar múltiples instancias de Matlab de forma automatizada, con el consumo de memoria que ello conlleva. Por ello, no se trata de una solución satisfactoria a este problema si lo que se busca es optimizar e incrementar la eficiencia del software desarrollado.

La segunda opción implica tener un gran conocimiento acerca de la programación en Java y en la compilación de sus clases. Además, genera un nuevo problema, la necesidad de sincronizar los hilos de ejecución creados y controlados por Java con el hilo principal de Matlab, lo cual no es ni sencillo ni seguro, ya que no está documentado oficialmente. Por otra parte, esta opción podría ser una solución si los hilos de ejecución programados en Java realizasen operaciones independientes a las rutinas del hilo principal controlado por Matlab.

4. Pobre gestión de la memoria.

El manejo de la memoria que hace Matlab, presenta un grave problema debido a que no tiene un mecanismo implícito de gestión de memoria como un *Garbage Collector* [25] y únicamente cuenta con el comando *clear* [26].

Esta característica se hace especialmente evidente cuando se trata de gestionar las figuras en las que se grafican los datos. El principal inconveniente es la no reasignación de la memoria automáticamente cuando se elimina o cierra una figura, lo que puede causar que Matlab se quede sin memoria si se están generando figuras en un bucle iterativo. Esta situación hace saltar un mensaje de excepción en la ejecución que bloquea el software y obliga al desarrollador a reiniciarlo. En la interfaz desarrollada no se presenta como un problema grave, ya que no se generan figuras y en el bucle únicamente se actualiza una misma figura. Sin embargo, es un aspecto a tener muy en cuenta.

5. Software privativo y necesidad de realizar las medidas en lugares con conexión a internet.

El IDE de Matlab precisa de una licencia y, dado el carácter de la licencia utilizada en el laboratorio, no puede utilizarse sin conexión a Internet. Esto implica que no pueden realizarse medidas en lugares en los que no haya una toma de red, como por ejemplo la cámara climática.

8. Conclusiones

Una vez finalizado el proceso de elaboración del bloque descriptivo del presente documento y finalizado el periodo de desarrollo del software, se concluye que:

1. Se ha implementado una interfaz gráfica para el control de analizadores de espectros ópticos en Matlab.
2. Se ha ampliado la funcionalidad del programa utilizado en el laboratorio, que únicamente constaba de una serie de funciones elementales.
3. Se ha logrado una adaptación satisfactoria de los recursos que ofrece Matlab a las necesidades propuestas facilitando las tareas de configuración, medida y visualización de los datos obtenidos mediante un analizador de espectros ópticos.
4. El software desarrollado es capaz de gestionar la adquisición de datos mediante dos tipos de instrumentos distintos.
5. Se ha mantenido una intención de universalidad en el proceso de desarrollo siendo posible tanto añadir nuevos parámetros de configuración como la inclusión de nuevos instrumentos de medida con pequeñas modificaciones del software.

9. Líneas futuras.

9.1 Ampliación de la interfaz a un nuevo analizador.

La primera opción de continuar el desarrollo de este trabajo consiste en ampliarlo mediante la configuración de la interfaz para operar con un nuevo OSA. Tal y como se ha especificado en las conclusiones, la ampliación de la interfaz implica la modificación de su código fuente y sería necesario estudiar detenidamente las nuevas necesidades que se desean satisfacer.

Una vez especificadas dichas necesidades, en primer lugar, se discutirá la inclusión de nuevos parámetros de configuración. La estructura del programa permite al usuario añadir nuevos parámetros de control en la ventana de configuración. Para ello, además de modificar la figura "set_parameters.fig" que define dicha ventana mediante la herramienta guide y añadir los campos correspondientes al vector de parámetros "param" en el código que controla dicha figura ("set_parameters.m"), es necesario actualizar la función "modelo_setup.m" del analizador para añadir las instrucciones SCPI precisas que configuren el OSA.

En segundo lugar, el proceso de configuración de la interfaz para la inclusión de un nuevo modelo consta de dos partes. La primera de ellas, es desarrollar el bloque de funciones propias del analizador dentro de un sub-directorio en la carpeta de la interfaz. Este bloque se compone de las funciones "modelo_init.m", "modelo_setup.m", "modelo_measure.m", "modelo_start.m" y "set_parameters.m" siguiendo la estructura planteada en el trabajo. La segunda parte consiste en añadir dicho modelo en las opciones de las estructuras de tipo *switch* de las funciones comunes "initialize.m" y "referenciador.m".

9.2 Desarrollo de una interfaz en Python 3.4 mediante el uso de la biblioteca Qt.

9.2.1 Introducción

Enlazando con la sección *7. Problemáticas encontradas en el uso de Matlab como entorno de desarrollo*, se propone la utilización del lenguaje interpretado Python 3.4 como alternativa para el desarrollo de una interfaz que, además de satisfacer todas las necesidades de la interfaz ya desarrollada, ofrezca una solución a las dificultades y problemáticas descritas en dicha sección y permita una sencilla ampliación del software a nuevos aparatos.

El lenguaje Python 3.4 cumple todas estas características ya que:

1. Se basa en el paradigma de la programación orientada a objetos.
2. Es un lenguaje interpretado.
3. Consta de un módulo de comunicación que permite el manejo y detección de buses GPIB (módulo *PyVISA*) y que incluye un potente administrador de recursos con el que se soluciona el problema de obtención de direcciones [27].
4. Dispone de la colección de software de computación científica *SciPy* que permite trabajar con matrices n-dimensionales mediante el paquete *Numpy* así como producir figuras de alta calidad haciendo uso de la librería *Matplotlib* [28] [29].
5. Cuenta con múltiples paquetes para el desarrollo objetos gráficos e interfaces de usuario.
6. Es un lenguaje de programación abierto, muy bien documentado y con una gran comunidad de desarrolladores. No precisa una licencia privativa y puede utilizarse sin problema en lugares en los que no haya conexión a la red.

Como una pequeña introducción a esta línea de desarrollo, se ha desarrollado el software que permite establecer una comunicación, configurar y realizar medidas en los analizadores utilizados para el desarrollo de la interfaz en Matlab. (AGILENT 86142A y EXFO). Por ello, se procederá a continuación a realizar una pequeña descripción del trabajo realizado en torno a esta línea.

9.2.2 Preparación del entorno de trabajo. Instalación del software necesario para comenzar el desarrollo.

En primer lugar, y con el objetivo de garantizar que se cumplen las nuevas necesidades propuestas, el entorno de trabajo se preparará en un ordenador personal. Así, antes de poder comenzar el desarrollo de la interfaz en Python 3.4, se hace necesario configurar este entorno de desarrollo correctamente. Para ello hay que instalar distintos paquetes de software, comenzando por el lenguaje de programación. Dentro de las versiones estables del lenguaje Python, se ha escogido la versión 3.4 por su compatibilidad con la librería PyQt4, que se empleará para la creación de las ventanas gráficas [30]. De esta forma, además de instalar el lenguaje Python 3.4, es necesario instalar las librerías *PyQt4* y *Matplotlib*, así como los paquetes *Numpy* y *PyVISA*.

Por otro lado, para el desarrollo de código se recomienda la instalación de un IDE, configurado para trabajar con Python. Entre todas las alternativas disponibles se escoge *PyCharm*, un potente IDE desarrollado por *JetBrains* para la programación en Python y que provee al desarrollador de un entorno que cuenta con todas las herramientas necesarias incluyendo un analizador de código y un depurador gráfico [31].

También es necesario instalar los drivers del adaptador GPIB-USB de Agilent utilizado en el laboratorio. Estos drivers son mantenidos actualmente por *Keysigth* y se encuentran disponibles para su descarga en su página web [32]. Una vez configurado todo correctamente, es posible comenzar el desarrollo de esta interfaz.

9.2.3 Estructura del código y jerarquización de clases.

El software que se va a comenzar a desarrollar seguirá la estructura del programa basado en una arquitectura funcional, desarrollado en Matlab y que ya se ha documentado. Por ello, constará de los tres bloques imprescindibles: comunicación, tratamiento de datos y visualización.

Sin embargo, se utilizará una programación orientada a objetos y una jerarquización de paquetes y distintos módulos que serán controlados por un módulo principal que definirá una ventana con la que interactuará el usuario. Así, el objetivo de este desarrollo realizado tras la conclusión del proyecto ya documentado, es demostrar la potencia y versatilidad del uso del lenguaje interpretado Python para la adquisición de datos.

El código se ha estructurado en un directorio principal en el que se localizan los módulos principales del programa y un subdirectorio "Configuraciones" que contiene los paquetes con los módulos desarrollados para cada analizador configurado para funcionar con esta interfaz.

Los módulos principales que se han desarrollado son:

1. "communicate.py": se encarga de inicializar y mantener la comunicación con el bus GPIB y contiene las clases necesarias para crear el objeto de comunicación.
2. "main_config.py": contiene la definición de la clase "_OSA" que almacena los distintos OSA para los que se ha configurado el software y maneja los archivos de configuración de cada analizador guardados en el directorio *Configuraciones*.
3. "main.py": define las funciones que hacen uso de las clases proporcionadas por los módulos de comunicación y configuración y operan con ellas para realizar las medidas.
4. "PyOSAUI.py": es el archivo principal del software y define y controla la interfaz gráfica de usuario.

Por otra parte, los paquetes propios de cada OSA almacenados en *Configuraciones*, contienen un solo módulo nombrado "modelo_config" que contiene tres clases "ModeloInit", "ModeloSetup" y "ModeloMeasure", donde *modelo* es la cadena de caracteres que identifica a cada analizador configurado. Estas tres clases contienen el método "self.comandos", que al ser ejecutado devuelve una lista con las instrucciones SCPI para la inicialización, configuración y medida respectivamente de cada OSA.

9.2.4 Establecimiento de la comunicación y utilización del módulo pyVISA mediante el módulo "communicate.py" desarrollado.

El módulo "communicate.py" importa el módulo pyVISA y consta de las clases *Initialize* e *Instr*. Tal y como se observa en el *Fragmento de código 31*, la clase *Initialize*, crea un objeto "ResourceManager" y almacena en la variable "lista" todas las comunicaciones que se encuentran disponibles. Además, cuenta con el método "show_concrete" que se ha desarrollado para, ante la introducción del parámetro "filter", almacenar en la variable "lista_filtrada" todas las comunicaciones del tipo que se especifique en dicho filtro. En la creación de un objeto de la clase

Initialize, el parámetro "filter" será la cadena de caracteres "GPIB", ya que es el tipo de comunicación que se va a utilizar. Sin embargo, la presencia del método "show_concrete" permite el uso de esta clase para crear el objeto de comunicación que se desee, siempre y cuando el tipo de bus de comunicaciones esté soportado por el módulo pyVISA [27].

La clase *Instr* se encarga de conectar el objeto creado al instrumento. Para ello, como se aprecia en el *Fragmento de código 32*, precisa en su inicialización de los parámetros "rm" e "instr_dir" que son el objeto "ResourceManager", que debe haberse creado previamente mediante el uso de la clase *Initialize* y la dirección GPIB seleccionada mediante el método "show_concrete" respectivamente.

Además, cuenta con los métodos "read_from_instrument" y "write_to_instrument" que se han desarrollado para permitir una comunicación bidireccional con el instrumento mediante el envío de las instrucciones SCPI correspondientes.

Fragmento de código [31]. Clase "Initialize".

```
class Initialize:

    def __init__(self):
        self.rm = visa.ResourceManager()
        self.lista=self.rm.list_resources()
    def show_concrete(self, filter):
        self.lista_filtrada=[]
        for item in self.lista:
            if filter in item:
                self.lista_filtrada.append(item)
```

Fragmento de código [32]. Clase "Instr".

```
class Instr:

    def __init__(self, rm, instr_dir):
        self.obj = rm.open_resource(instr_dir)
        self.output = []
    def read_from_instrument(self, instruction):
        self.output = self.obj.query_ascii_values(instruction)
```

```
        return self.output
    def write_to_instrument(self, instruction_list):
        for item in instruction_list:
            try:
                self.obj.write(item)
            except:
                break;
```

9.2.5 Configuración del OSA y manejo de las instrucciones SCPI.

Una vez definidas las clases necesarias para crear el objeto de tipo *Instr,* y desarrollados los métodos necesarios para lograr una comunicación satisfactoria con el analizador, es posible comenzar el desarrollo del software que permita al usuario configurar, medir y visualizar los datos obtenidos.

Para ello, lo primero que se debe realizar es la creación de los paquetes de configuración de cada OSA y el módulo "main_config.py" que maneje dichos paquetes. Estos paquetes contienen un único módulo que se describirá a continuación.

El módulo de configuración de cada analizador define tres clases que contienen las instrucciones SCPI necesarias para la inicialización, configuración y toma de medidas de cada analizador. Como puede observarse en los *Fragmentos de código 33, 34 y 35* en los que se muestran las clases desarrolladas para el modelo de analizador AGILENT 86142A, las tres clases cuentan con el método "comandos", que devuelven la lista de instrucciones SCPI que es necesario enviar al OSA para realizar las tres operaciones. El vector "param", al igual que en la interfaz desarrollada en Matlab, contiene los parámetros de configuración del analizador, y debe introducirse como argumento en la creación de los objetos definidos por estas clases.

El módulo "main_config.py" almacena los dispositivos para los que se ha configurado el software y es desde este archivo desde donde debe añadirse uno nuevo si así se desea. Para añadir la configuración de un nuevo OSA al programa, se deberá declarar en este archivo, así como incluir un paquete con el mismo identificador y su respectivo archivo de configuración "identificador_config.py" en el directorio "Configuraciones". Tal y como puede observarse en el *Fragmento de código 36*, la clase contiene una lista que almacena los identificadores y el método

"select", que al ser ejecutado devuelve la lista de parámetros que es necesario introducir para la configuración del OSA y las tres clases definidas en el archivo de configuración ya descrito en el párrafo anterior.

Fragmento de código [33]. Clase Osa86142aInit.

```
class Osa86142aInit:
    def __init__(self, param):
        self._command=[
            'Init:Cont On',
            'Sens:Swe:Poin %d' %param[1],
            'Disp:Wind:Trac:Stat TrA, On',
            'Trac:Feed:Cont TrA, Alw'
        ]
    def comandos(self):
        return self._command
```

Fragmento de código [34]. Clase Osa86142aSetup.

```
class Osa86142aSetup:
    def __init__(self, param):
        self._command=[
            'Sens:Bwid:Res %d nm' %param[2],
            'Disp:Wind:Trac:Y:Scal:Rlev %d dBm' %param[3],
            'Sens:Wav:Cent %d nm' %param[4],
            'Sens:Wav:span %d nm' %param[5],
            'DISP:WIND:TRAC:Y:SCAL:PDIV %d ' %param[6],
            'Sens:Pow:Dc:Rang:Low %d dBm' %param[7]
            ]
    def comandos(self):
        return self._command
```

Fragmento de código [35]. Clase Osa86142aMeasure.

```
class Osa86142aMeasure:
    def __init__(self, param=''):
        self._command='Trac:Data:Y? TrA'
    def comandos(self):
        return self._command
```

Fragmento de código [36]. Clase "_OSA" del módulo "main_config.py".

```
class _OSA:
    def __init__(self):
        self.param = []
        self.lista = [
            'Exfo',
            'Osa86142a'
        ]
    def select(self, arg):
        self.param.append(arg)
        if arg == 0:
            from Configuraciones import ExfoInit as Init,...
                  ExfoSetup as Setup, ExfoMeasure as Measure
            self.paramlen = 5
            self.paramlist = ['id',
                              'Limite inferior',
                              'Limite superior',
                              'Sensibilidad',
                              'Numero puntos'
                              ]
        elif arg == 1:
            from Configuraciones import Osa86142aInit as Init,...
                 Osa86142aSetup as Setup, Osa86142aMeasure as Measure
            self.paramlen = 8
            self.paramlist = ['id',
                              'Numero puntos',
                              'RBW',
                              'Nivel Referencia',
                              'Centro',
                              'Span',
                              'Escala',
                              'Sensibilidad'
                              ]
        else:
            return -1
        return Init, Setup, Measure
```

9.2.6 Desarrollo de las funciones necesarias para la creación de los objetos y manejo de los mismos.

El módulo “main.py” contiene las definiciones de las funciones necesarias para, mediante el uso de las clases ya definidas, iniciar la comunicación, conectarse al instrumento, configurarlo, medir y procesar los datos obtenidos. Estas funciones son las que utiliza la interfaz gráfica para operar con el OSA.

La función “init_connection” (*Fragmento de código 37*), nicializa la conexión con el modulo pyVISA y retorna las conexiones *GPIB* disponibles, así como los modelos osa para los que se ha configurado la interfaz.

La función “connect_to_instr” (*Fragmento de código 38*), establece la conexión con el OSA seleccionado y devuelve las clases necesarias para iniciar, configurar y medir con dicho OSA, así como el objeto de la clase *Instr* creado para la comunicación.

La función “instr_config” (*Fragmento de código 39*) hace uso de las clases obtenidas mediante la función “init_conection” y del vector de parámetros *param* para generar los objetos de inicialización, configuración y medida del analizador, inicializarlo y configurarlo y por último devolver las instrucciones de medida en la variable *osa_measure*.

Las funciones “measure_loop_MODELO” *Fragmentos de código 40 y 41*), utilizan las instrucciones proporcionadas por la función “instr_config” en la variable *osa_measure,* el objeto *instr* y el vector de parámetros *param* para, haciendo uso de la clase definida en el módulo “communicate.py” solicitar al analizador la medida para la que se ha configurado. Una vez obtenidos estos datos, la interfaz hará uso de las funciones “get_yplot” (*Fragmento de código 42*) y “savitzky_golay” (*Fragmento de código 43*) que, a partir de una medida y una referencia, suavizan los datos mediante este filtro (que sustituye a la función *smooth* utilizada en Matlab) y devuelven los vectores “x” e “y_plot” con la información preparada para ser visualizada [33] [34].

Fragmento de código [37]. Función "init_conection".

```
def init_connection():
    osa = _OSA()
    com = Initialize()
    com.show_concrete('GPIB')
    lista = com.lista_filtrada
    return com, lista, osa
```

Fragmento de código [38]. Función "connect_to_instr".

```
def connect_to_instr(com, lista, osa, arg, sel):
        instr = Instr(com.rm, lista[arg])
    [init, setup, measure] = osa.select(sel)
    return init, setup, measure, instr
```

Fragmento de código [39]. Función "instr_config".

```
def instr_config(init, setup, measure, instr, param):
    osa_init = init(param)
    instr.write_to_instrument(osa_init._command)
    osa_setup = setup(param)
    instr.write_to_instrument(osa_setup._command)
    osa_measure = measure(param)
    return osa_measure
```

Fragmento de código [40]. Función "measure_loop_agilent".

```
def measure_loop_agilent(instr, osa_measure, param):
    y = np.array(instr.read_from_instrument(osa_measure._command),
        dtype='float')
    x = np.linspace((param[4] - param[5] / 2), (param[4] + param[5] / 2),
        param[1])
    return x, y
```

Fragmento de código [41]. Función "measure_loop_exfo".

```
def measure_loop_exfo(instr, osa_measure, param):
    buffer_dict = {}
    instr.write_to_instrument(['INIT0:AUTO 1\n'])
    alm = (instr.read_from_instrument('INIT0:ACQ:STAT?\n'))
    while not int(alm[0]):
        alm = (instr.read_from_instrument('INIT0:ACQ:STAT?\n'))
        sleep(0.01)
    l = len(osa_measure._command)
    for i in range(0, l):
        buffer_dict['mes%d' % i] =
np.array(instr.read_from_instrument(osa_measure._command[i]), dtype='float')
    y = np.array(buffer_dict['mes0'], dtype='float')
    for i in range(1, l):
        y = np.append(y, buffer_dict['mes%d' % i])
    x = np.linspace(param[1], param[2], len(y))
    return x, y
```

Fragmento de código [42]. Función "get_yplot".

```
def get_yplot(y_ref, y_med):
    y = y_med - y_ref
    y_plot = savitzky_golay(y, 51, 3)
    return y_plot
```

Fragmento de código [43]. Función "savitzky_golay".

```
def savitzky_golay(y, window_size, order, deriv=0, rate=1):
        window_size = np.abs(np.int(window_size))
        order = np.abs(np.int(order))
        order_range = range(order + 1)
        half_window = (window_size - 1) // 2
        b = np.mat([[k ** i for i in order_range] for k in range(-half_window,
half_window + 1)])
        m = np.linalg.pinv(b).A[deriv] * rate ** deriv * factorial(deriv)
        firstvals = y[0] - np.abs(y[1:half_window + 1][::-1] - y[0])
        lastvals = y[-1] + np.abs(y[-half_window - 1:-1][::-1] - y[-1])
        y = np.concatenate((firstvals, y, lastvals))
        return np.convolve(m[::-1], y, mode='valid')
```

9.2.7 Desarrollo de la interfaz y manejo de las funciones definidas en el módulo "main.py".

El último paso de desarrollo consiste en la creación de una interfaz que permita al usuario el manejo de las funciones definidas en el módulo "main.py" y la visualización de los datos, tal y como se ha hecho con Matlab. La interfaz que se ha comenzado a desarrollar, se ha realizado mediante el manejo de las librerías *PyQt4* y *Matplotlib* que facilitan respectivamente un entorno de desarrollo de GUIs y la posibilidad de elaborar figuras de representación.

El módulo "pyOSAUI.py" define dicha ventana, generada por la herramienta QtDesigner y almacenada en el archivo "main.ui" y se sirve de la ventana "dialogo_config.ui" para dar soporte a los menús de configuración. Esta ventana consta de los espacios para poder realizar la visualización de los datos y tres botones de control, que permiten al usuario establecer una conexión, configurar el OSA para realizar la medida y medir respectivamente. Además, cuenta con una consola en modo lectura para transmitir información al usuario.

Para la elaboración del mapa de colores, tal y como se hace en la interfaz desarrollada con Matlab con la instrucción *pcolor*, se utiliza la función *pcolormesh* incluida en el módulo "pyplot" dentro de la librería *Matplotlib* y cuya sintaxis es idéntica a la utilizada por Matlab.

El funcionamiento de esta interfaz, a diferencia de la interfaz desarrollada mediante Matlab, se basa en el uso de la gran capacidad de gestión de múltiples hilos de ejecución que ofrece Python. Así, se utiliza un hilo de ejecución paralelo para controlar el bucle de medidas y permitir al usuario interactuar libremente con la ventana (por ejemplo, moverla libremente por la pantalla o pausar/parar la medida sin retardos) e incluso interactuar con los datos visualizados en las figuras. De esta forma, es posible hacer zoom en las medidas que se están mostrando, cambiar los límites de los ejes, modificar los colores de los datos que se muestran o exportar una medida como imagen sin necesidad de detener la obtención de datos.

Debido a que la documentación de este desarrollo no es uno de los objetivos del trabajo que se ha elaborado y a la extensión que precisaría una correcta descripción de esta interfaz, se ha incluido el código comentado en la sección *11.3. Código correspondiente a la función principal para la interfaz comenzada a desarrollar en Python 3.4* de los anexos.

9.2.8 Muestra del funcionamiento de la interfaz desarrollada en Python 3.4

Tras el desarrollo realizado en torno a esta línea, se procedió a la utilización del ordenador personal en el que se ha configurado el entorno de desarrollo para realizar una medición en ambos analizadores, obteniendo unos resultados más que satisfactorios. En la *Figura 31* se observa la ventana principal de la interfaz desarrollada inmediatamente después de la ejecución del software mientras que en la *Figura 32* se muestra el programa en funcionamiento.

Estas figuras demuestran la potencia y las posibilidades que ofrece el desarrollo de un software de adquisición de datos directamente desde un lenguaje de propósito general basado en el paradigma de la programación orientada a objetos.

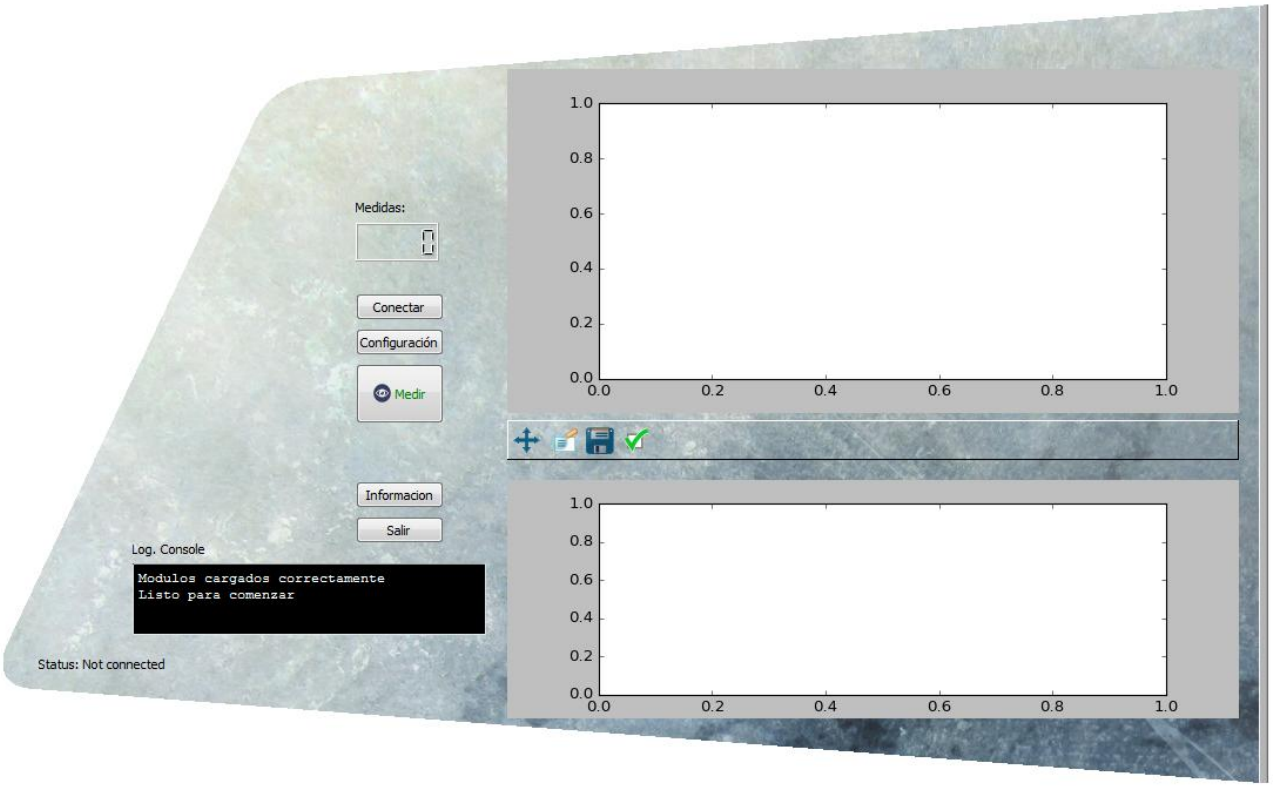

Figura 31. Ventana principal PyOSAUI.

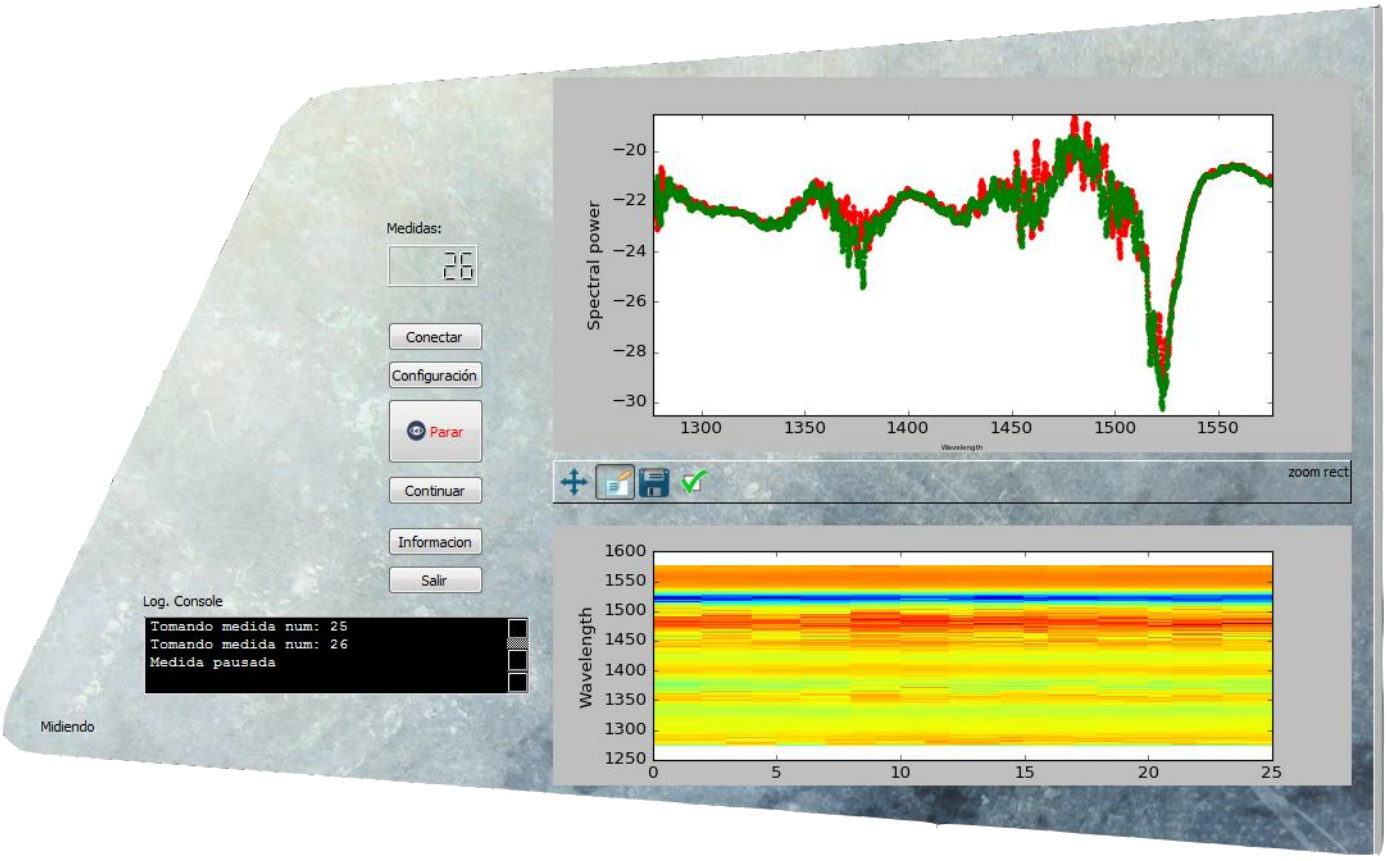

Figura 32. Software de adquisición en funcionamiento.

9.3 Guardado de datos en un servidor

Por último, se propone la utilización de un modelo de software cliente-servidor para el almacenamiento de los datos obtenidos en otro pc, con el doble objetivo de disminuir la pérdida de información en caso de un fallo del dispositivo de control y facilitar el acceso a estos datos desde cualquier equipo.

10. Bibliografía y referencias.

[1] http://www.hp9845.net/9845/tutorials/hpib/

[2] Keysight Command reference – Volume 1.

[3] SCPI – 99. Volume 1: Syntax and Style. Introduction: SCPI goals.

[4] Agrawal G. P. Fiber – Optic Communication Systems, 3ed, Wiley – Interscience, EUA, 2002.

[5] Thorlabs OSA 202 User Manual, pag. 8: 4.13 Resolution and Sensitivity.

[6] Thorlabs CCS175 User Manual, pag. 66.

[7] Manual OceanOptics Flame-NIR pag. 4.

[8] Thorlabs OSA 202 User Guide, Spectral Resolution.

[9] Matlab documentation. Instrument Control Toolbox. GPIB.

[10] Labview User Manual. Virtual Instrument Software Architecture.

[11] Agilent 86140A Optical Spectrum Analyzer Family Technical Specifications pag. 2.

[12] Handbook of Interferometers. Chapter 19. Ignacio Del Villar, Ignacio R. Matías, Francisco J. Arregui. Long-Period-Fiber-Grating-Based Interferometers. Introduction.

[13] National Instruments. Measurement & Automation Explorer (MAX).

[14] Matlab documentation. Structure Array.

[15] Matlab documentation. Programatic workflow.

[16] Matlab documentation. Guide-UI design environment.

[17] Matlab documentation. Callbacks for different user actions.

[18] Matlab documentation. Guidata.

[19] Matlab documentation. Construct function handle from string.

[20] Matlab documentation. Uicontrol, uiwait.

[21] Matlab documentation. File operations.

[22] Matlab documentation. Instrument Control Toolbox. Write text to instrument.

[23] Matlab documentation. Curve Fitting Toolbox. Smooth response data.

[24] Matlab documentation. Try, catch and MException Class.

[25] www.xpode.com. Memory allocation, garbage collection, overflow and underflow.

[26] Matlab documentation. Clear command.

[27] PyVISA documentation.

[28] SciPy documentation. Numpy array.

[29] Matplotlib documentation. Pyplot.

[30] PyQt4 Reference guide.

[31] JetBrains, PyCharm Docs & Demos.

[32] Keysight Instrument Drivers.

[33] IEEE Signal Processing Magazine. “What is a Savitzky-Golay Filter?” Ronald W. Schafer.

[34] SciPy documentation. Savitzky-Golay Filtering.

11. Anexos

11.1 Instrucciones SCPI, para la inicialización, configuración y medida necesarias para los modelos de OSA configurados.

Se citan las instrucciones SCPI utilizadas, obtenidas de forma literal del programa heredado. Con el fin de clarificar el cometido de cada comando, se han incluido los parámetros de configuración en forma de variables y se muestran en mayúsculas con el objetivo de diferenciarlos de dichas instrucciones SCPI.

Analizador EXFO.

- Inicialización.

```
Calc0:Tran                    # Inicia la test de transmitancia
Calc0:Tran:Sel 1,2            # Selecciona las trazas 1 y 2
Inst0:Nsel 2                  # Marca como activa la traza 2
```

- Configuración.

```
Sens0:Wav:Rang INF,SUP\n      # Establece el rango de medida
Sens0:Pow:Rang SENS           # Establece la sensibilidad
```

- Medida.

```
Init0:Auto 1\n          # Comienza la obtención de datos en modo automático
Trac0:Data? NUMBER \n         # Solicitud de datos
```

Analizador AGILENT 86142A.

- Inicialización.

```
Init:Cont On                        # Inicializa el analizador
Disp:Wind:Trac:Stat TrA, On         # Activa la traza A
Trac:Feed:Cont TrA, Alw  # Indica al analizador que siempre actualice la traza A
```

- Configuración.

```
Sens:Swe:Poin POINTS                          # Fija el numero de puntos
Sens:Bwid:Res RBW nm                          # Configura la resolución
Disp:Wind:Trac:Y:Scal:Rlev REF_LEVEL dBm      # Nivel de la referencia
Sens:Wav:Cent CENTER nm                       # Centro de la medida
Sens:Wav:span SPAN nm                         # Rango de span
DISP:WIND:TRAC:Y:SCAL:PDIV SCALE              # Escala de la medida
Sens:Pow:Dc:Rang:Low SENS dBm                 # Sensibilidad
```

- Medida.

```
Trac:Data:Y? TrA                          # Solicitud de datos de la traza A
```

11.2 Desglose de la estructura de variables "data".

Se citan los campos de la estructura "data" clasificándolos según su funcionalidad.

- Variables de control de flujo del programa:

  data.control: indica si el bucle de medidas debe continuar iterando.

  data.ref_control: almacena el tipo de gestión de referencias.

  data.loop: indica al software si está activo el modo bucle o si es necesario configurar el OSA.

  data.zoom: indica si el usuario ha elegido realizar zoom en las medidas.

  data.error: almacena los identificadores de los errores encontrados.

  data.primary_path: almacena la ruta del directorio principal del software.

  data.path: almacena la ruta del directorio correspondiente al OSA seleccionado.

  data.cont: almacena el índice de la medida que se está realizando.

  data.last: almacena el índice correspondiente a la última medida programada.

  data.ref_num: almacena el número de referencias cargadas en memoria.

- Variables de configuración:

  data.osa: almacena modelo OSA.

  data.param: almacena los valores de los parámetros de configuración del OSA.

  data.zoom_level: almacena el numero de tramos para el zoom.

  data.smooth: almacena el nivel de suavizado de las curvas.

data.puntos: almacena el numero de puntos de una referencai cargada en memoria.

data.zzN: almacena los valores de los parámetros "centro" y "span" en el caso de que se vaya a realizar un zoom.

data.last_rbwN: almacena el valor del parámetro "RBW" de la medida N para el analizador AGILENT 86142A, en el caso de que haya varias medidas programadas.

data.last_refN: almacena el valor del parámetro "nivel de referencia" de la medida N para el analizador AGILENT 86142A, en el caso de que haya varias medidas programadas.

data.last_scaleN: almacena el valor del parámetro "Escala" de la medida N para el analizador AGILENT 86142A, en el caso de que haya varias medidas programadas.

data.last_sensN: almacena el valor del parámetro "Sensibilidad" de la medida N para el analizador AGILENT 86142A, en el caso de que haya varias medidas programadas.

data.last_centerN: almacena el valor del parámetro "Centro" de la medida N para el analizador AGILENT 86142A, en el caso de que haya varias medidas programadas.

data.last_spanN: almacena el valor del parámetro "span" de la medida N para el analizador AGILENT 86142A, en el caso de que haya varias medidas programadas.

data.inf: almacena el límite inferior del rango de medida para el OSA EXFO.

data.sup: almacena el límite superior del rango de medida para el OSA EXFO.

data.sens: almacena el valor del parámetro "Sensibilidad" para el OSA EXFO.

- Variables de almacenamiento de datos:

  data.obj: almacena el objeto de comunicación.

  data.N: almacena el numero de puntos solicitado por el software al analizador.

  data.A: almacena todas las medidas realizadas desde que se pulsa el botón comenzar.

  data.x: almacena el vector de longitud de onda de la medida que se va a visualizar en pantalla.

  data.x_reference: almacena el vector de longitudes de onda de referencia.

  data.y_reference: almacena el vector de potencia espectral de referencia.

  data.x_persistent: almacena el vector de longitudes de onda correspondiente a la primera iteración del bucle de medidas.

  data.y_persistent: almacena el vector de potencia espectral correspondiente a la primera iteración del bucle de medidas.

  data.time_str: almacena hora de captura.

  data.id: almacena identificador de archivo.

data.x_med: almacena el último vector de longitudes de onda recibido para zoom.

data.y_med: almacena el último vector de potencia espectral recibido para zoom.

data.y_save: almacena las potencias espectrales para zoom.

data.x_refN: almacena el vector de longitud de onda de referencia de la medida N en el caso de que se vaya a realizar un zoom.

data.y_refN: almacena el vector de potencia espectral de referencia de la medida N en el caso de que se vaya a realizar un zoom.

11.3 Código correspondiente a la función principal para la interfaz comenzada a desarrollar en Python 3.4.

```
'''
*****************************************************************************************
***                                                                                   ***
***                                                                                   ***
***                          .d88888b.   .d8888b.        d8888 888     888 8888888     ***
***                         d88P" "Y88b d88P  Y88b      d88888 888     888   888       ***
***                         888     888 Y88b.          d88P888 888     888   888       ***
***         88888b.  888  888 888     888  "Y888b.      d88P 888 888     888   888       ***
***         888 "88b 888  888 888     888     "Y88b.   d88P  888 888     888   888       ***
***         888  888 888  888 888     888       "888  d88P   888 888     888   888       ***
***         888 d88P Y88b 888 Y88b. .d88P Y88b  d88P d8888888888 Y88b. .d88P   888       ***
***         88888P"   "Y88888  "Y88888P"   "Y8888P" d88P     888  "Y88888P"  8888888     ***
***         888           888                                                         ***
***         888      Y8b d88P                                                         ***
***         888       "Y88P"                                                          ***
***                                                  jon.gmz.lbat@gmail.com           ***
***                                                                                   ***
*****************************************************************************************
'''

from PyQt4.uic import loadUiType
from PyQt4 import QtGui, QtCore
from matplotlib.figure import Figure
from matplotlib.backends.backend_qt4agg import (
    FigureCanvasQTAgg as FigureCanvas)
from matplotlib.backends.backend_qt4 import NavigationToolbar2QT
from time import strftime
import sys, os, glob, main


Ui_MainWindow, QMainWindow = loadUiType('main.ui')       # Carga clases de la ventana principal
Ui_Dialog, QDialog = loadUiType('dialogo_config.ui')     # Carga clases para la ventana de dialogo


class MeasureThread(QtCore.QThread):
    '''Define un hilo de ejecución dedicado al bucle de medidas que mide, plotea y guarda las
medidas'''
    def __init__(self):
        '''Inicialización de la clase define la señal del thread y declara las variables
necesarias'''
        QtCore.QThread.__init__(self)
        self.signal = QtCore.SIGNAL('signal')
        self.measure_dict = {}  # Almacen medidas
        self.measure_c = 0      # Contador medidas
        self.x_med = 0          # Almacena longitudes de onda
        self.y_plot = 0         # Almacena potencia espectral
        self.x_persistent = 0   # Almacena la primera medida
```

```
        self.y_persistent = 0   # Almacena la primera medida
        self.x_bird = 0         # Almacena el rango de numero de medidas para el mapa de colores
        self.x_bird_vector = 0  # Almacena el vector longitudes de onda para el mapa de colores
        self.y_bird_matrix = 0  # Almacena la matriz con todas las potencias espectrales

    def __del__(self):
        self.wait()

    def measure(self):
        '''Solicita los datos al instrumento'''
        text = 'Tomando medida num: ' + str(self.measure_c) + '\n'
        window.log_send.emit_log(text)
        self.x_med, self.y_med = window.return_measure()           # Solicita la medida
        self.y_plot = main.get_yplot(window.y_ref, self.y_med)     # Solicitud de procesado de datos

    def save_measure(self):
        '''Guarda los datos obtenidos en archivos .txt en el directorio medidas con un subdirectorio
de fecha'''
        window.get_index(0)                                         # Obtiene identificador
        self.name_matrix = str(window.name_index) + ' matrix.txt'   # Genera nombre archivos
        self.name_wavelength = str(window.name_index) + ' wavelength.txt'
        if self.measure_c == 0:
            self.y_bird_matrix = self.y_plot
            self.x_persistent = self.x_med
            self.y_persistent = self.y_plot
            self.measure_dict['x'] = self.x_med
            self.measure_dict['y0'] = self.y_plot
        else:
            self.measure_dict['y%d', self.measure_c] = self.y_plot
            self.y_bird_matrix = main.np.vstack((self.y_bird_matrix,  # Añade el nuevo vector
                                                 self.y_plot))
        main.np.savetxt(self.name_matrix,                           # Guarda matriz de potencias
                        self.y_bird_matrix,
                        delimiter=' ')
        main.np.savetxt(self.name_wavelength, self.x_med, delimiter=' ') # Guarda longitudes de onda

    def plot_birdview(self):
        '''Plotea la vista de pajaro a partir de un diccionario con las medidas realizadas y el
numero de medidas que contiene'''
        self.x_bird = main.np.arange(self.measure_c)          # Prepara rango de numero de medidas
        self.x_bird_vector = self.x_med                       # Obtiene vector de longitudes de onda
        window.birdview.set_xlim([0, self.measure_c-1])       # Configura la ventana de medida
        window.birdview.set_ylabel('Wavelength (nm)')
        window.birdview.pcolormesh(self.x_bird, self.x_bird_vector,
                                   self.y_bird_matrix.T)      # Genera mapa de colores
        window.canvas_birdview.draw()                         # Muestra el mapa de colores

    def plot(self):
        '''Grafica los datos obtenidos'''
        window.spectral_power.set_xlabel('Wavelength (nm)',
                                         fontsize=5)          # Configura ventana de medida
```

```
        window.spectral_power.set_ylabel('Spectral power (dB)')
        window.spectral_power.set_xlim([window.x_lim_inf, window.x_lim_sup])
        window.spectral_power.plot(self.x_med, self.y_plot, 'r.')               # Genera la gráfica
        window.spectral_power.plot(self.x_persistent, self.y_persistent, 'g.')
        window.canvas_measure.draw()                                            # Muestra la gráfica

    def run(self):
        '''Instrucciones a ejecutar para el thread de medida'''
        while window.measure_control == 1:                  # Comprueba condición del bucle
            if not window.pausa:
                window.lcd.display(self.measure_c)          # Actualiza el contador de medidas
                window.spectral_power.clear()               # Vacia las ventanas de visualización
                window.birdview.clear()
                self.measure()                              # Solicita la medida
                self.plot()                                 # Dibuja la medida obtenida
                if self.measure_c > 1:
                    self.plot_birdview()                    # Dibuja el mapa de colores
                self.save_measure()                         # Almacena la medida realizada
                self.measure_c += 1                         # Aumenta el contador de medidas
            else:
                main.sleep(0.01)                            # Bucle en pausa


class ReferenceThread(QtCore.QThread):
    '''Define el hilo de ejecución dedicado a la tima de referencias'''
    def __init__(self):
        '''Inicialización de la clase, genera la señal del thread'''
        QtCore.QThread.__init__(self)
        self.signal = QtCore.SIGNAL('signal')
        text = 'Referenciando..'                            # Actualiza consola de logs
        window.log_send.emit_log(text)

    def __del__(self):
        self.wait()

    def run(self):
        '''Toma la referencia, y actualiza la variable de control de flujo window.ref_wait'''
        window.x_ref, window.y_ref = window.return_measure()    # Solicita una medida
        window.ref_wait = 0


class ConnectDialog(QDialog, Ui_Dialog):
    '''Define un diálogo para la conexión al instrumento'''
    def __init__(self, com, lista, osa):
        '''Conecta las señales de los botones a sus slots e inicializa las variables'''
        super(ConnectDialog, self).__init__()
        self.setupUi(self)
        self.setWindowTitle('Conectar')             # Configura el titulo de la ventana
        self.button_aceptar.clicked.connect(self.button_aceptar_clicked)    # Conecta señales
        self.button_cancelar.clicked.connect(self.button_cancelar_clicked)
        self.indicador = 0                          # Indica menús superados por el usuario
```

```
        self.osa = osa                                  # Almacena clases osa
        self.com = com                                  # Almacena objeto de comunicación
        self.lista = lista                              # Almacena lista de modelos

    def button_aceptar_clicked(self):
        '''Función slot para la señal del botón de aceptar'''
        if self.indicador == 0:
            self.conn_selected = self.lista_principal.currentRow()  # Captura conexión seleccionada
            self.lista_principal.clear()                            # Borra los items de la lista
            self.text_cabecera.setText('Seleccione OSA: ') # Cambia el título para el siguiente menú
            for item in self.osa.lista:
                self.lista_principal.addItem(item)                  # Muestra los osas configurados
            self.indicador = 1                                      # Cambia la variable indicadora
            text = 'Conexión fijada\n'                              # Actualiza consola de logs.
            window.logoutput(text, 0)
        else:
            text = 'Osa fijado\n'                                   # Actualiza consola de logs
            window.logoutput(text, 0)
            self.button_aceptar.setEnabled(False)                   # Desactiva botones
            self.button_cancelar.setEnabled(False)
            self.osa_selected = self.lista_principal.currentRow()   # Captura osa seleccionado
            self.lista_principal.clear()
            self.close()
            text = 'Iniciando conexión\nConstruyendo objeto de comunicación\n'
            window.logoutput(text, 0)                               # Actualiza consola de logs
            '''Carga las clases necesarias para la inicialización, configuración y medida así como
el objeto de comunicación instr'''
            self.init, self.setup, self.measure, self.instr = main.connect_to_instr(self.com,
               self.lista,self.osa, self.conn_selected, self.osa_selected)
            window.status = 1                                       # Actualiza estado del software
            window.text_status.setText('Status: Connected to ' + self.lista[self.conn_selected] +
                                       '; Using ' + self.osa.lista[self.osa_selected])
            text = 'Conexión realizada con éxito\n'                 # Actualiza consola de logs
            window.logoutput(text, 0)

    def button_cancelar_clicked(self):
        '''Función slot para la señal del botón de cancelar'''
        self.close()        # Cierra la ventana de conexión


class ConfigDialog(QDialog, Ui_Dialog):
    '''Define el diálogo para la configuración de una medida'''
    def __init__(self):
        '''Conecta las señales con los slots correspondientes y actualiza la lista de parámetros a
configurar'''
        super(ConfigDialog, self).__init__()
        self.setupUi(self)
        self.setWindowTitle('Configuración de la medida')                   # Título de la ventana
        self.button_aceptar.clicked.connect(self.button_aceptar_clicked)    # Conecta señales de los
botones
        self.button_cancelar.clicked.connect(self.button_cancelar_clicked)
```

```
        self.lista_principal.itemDoubleClicked.connect(self.modify_item)
        self.text_cabecera.setText('Parámetros de medida. Doble click para modificar:')
        if window.init and window.connect_dialog.osa_selected == 0:
            self.param = [None, 1250, 1600, 3, 10000]                # Valores predeterminados EXFO
            for i in range(0, window.osa.paramlen - 1):
                self.lista_principal.addItem(window.osa.paramlist[i + 1] + ': ' + str(self.param[i +
1]))
        else:
            self.param = []
            self.param.append(None)
            if window.status == 2:
                for i in range(0, window.osa.paramlen - 1):
                    self.param.append(window.param[i+1])
                    self.lista_principal.addItem(window.osa.paramlist[i + 1] + ': ' +
str(self.param[i+1]))
            else:
                for i in range(0, window.osa.paramlen - 1):
                    self.lista_principal.addItem(window.osa.paramlist[i + 1] + ': ')
                self.param = [None] * window.osa.paramlen
        self.param[0] = 0

    def button_aceptar_clicked(self):
        '''Llama a la funcion instr_config definida en el modulo main para configurar el OSA'''
        param_filt = [x for x in self.param if x is not None]
        if len(self.param) == len(param_filt):      # Comprueba que no se hayan olvidado valores
            text = 'Configurando OSA..\n'
            window.logoutput(text, 0)               # Actualiza consola de logs si todo es correcto
            self.close()
            '''Configura el OSA y devuelve instrucciones de medida'''
            self.osa_measure = main.instr_config(window.connect_dialog.init,
window.connect_dialog.setup,
                                                 window.connect_dialog.measure,
window.connect_dialog.instr, self.param)
            text = 'OSA configurado con éxito\n'
            window.logoutput(text, 0)                       # Actualiza consola de logs
            text = window.text_status.text()
            window.text_status.setText(text + '. Ready to measure') # Actualiza estado del programa
            window.status = 2
            window.param = self.param
        else:
            text = 'Error: Falta información\n'         # Aviso si faltan valores de configuración
            window.logoutput(text, 0)
            msg_box = QtGui.QMessageBox()
            msg_box.setWindowTitle('Error')
            msg_box.setText('Faltan parámetros por introducir')
            msg_box.exec_()

    def button_cancelar_clicked(self):
        '''Slot para el boton cancelar'''
        self.close()                                        # Cierra ventana de configuración
```

```
    def modify_item(self):
        '''Slot para el doble click en un elemento de la lista'''
        while True:
            dialog = QtGui.QInputDialog()                    # Configura ventana
            dialog.setWindowTitle('Edición parámetros')
            dialog.setLabelText('Introduzca valor del parámetro: "' +
self.lista_principal.currentItem().text() + '"')
            dialog.setCancelButtonText('Cancelar')
            dialog.setOkButtonText('Aceptar')
            dialog.setInputMode(0)
            ok = dialog.exec_()
            if ok:
                try:
                    row = self.lista_principal.currentRow()      # Modifica parámetro seleccionado
                    a = int(dialog.textValue())
                    self.lista_principal.currentItem().setText(window.osa.paramlist[row+1] + ': ' +
str(a))
                    self.param[row+1] = a
                    text = 'Fijado: ' + self.lista_principal.currentItem().text() + '\n'
                    window.logoutput(text, 0)
                    window.init = 0
                    break
                except ValueError:                          # Detecta si no se ha introducido un número
                    text = 'Error en parámetro: ' + self.lista_principal.currentItem().text() + '\n'
                    window.logoutput(text, 0)
                    msg_box = QtGui.QMessageBox()
                    msg_box.setText('Introduzca un valor numérico.')
                    msg_box.exec_()
            else:
                break


class NavigationToolbar(NavigationToolbar2QT):
    '''Define la barra de opciones de la figura de ploteo de potencia espectral'''
    toolitems = [t for t in NavigationToolbar2QT.toolitems if      # Barra de opciones personalizada
                 t[0] in ('Pan', 'Zoom', 'Save')]
    def __init__(self, canvas, parent):
        super(NavigationToolbar, self).__init__(canvas, parent)
        actions = self.findChildren(QtGui.QAction)
        for a in actions:
            if a.text() == 'Customize':
                self.removeAction(a)
                break
        NavigationToolbar2QT.__init__(self, canvas, parent)


class LogSignal(QtCore.QObject):
    '''Define la señal para poder actualizar la consola de logs desde los threads secundarios '''
    print_log = QtCore.pyqtSignal(str)

    def __init__(self):
```

```
        QtCore.QObject.__init__(self)

    def emit_log(self, log_text):
        self.print_log.emit(log_text)


@QtCore.pyqtSlot(str)
def set_log(log_text):
    '''Define el slot para la consola de logs'''
    window.logoutput(log_text, 0)


class MainWindow(QMainWindow, Ui_MainWindow):
    '''*****************************************************************************************
    **Ventana principal de la interfaz.                                                     **
    *****************************************************************************************
    **Se trata de la clase principal de este programa, contiene los métodos necesarios para **
    **1) Responder ante las señales de pulsado de los botones.                              **
    **2) Permitir al usuario arrastrar la ventana.
    **3) Gestionar adecuadamente los hilos paralelos y todos los eventos de ejecución       **
    *****************************************************************************************'''

    def __init__(self,):
        '''*************************************************************************************
        **Inicialización de la clase                                                         **
        *************************************************************************************
        **1) Se crean los objetos com, lista y osa a partir de la función "init_connection"     **
        **2) Se conectan las señales de los botones a las funciones correspondientes.           **
        **3) Se generan las figuras que contendran las gráficas.                               **
        **4) Se cargan las imágenes "back" y "wall" que se utilizan para dar forma y fondo a la **
        **   ventana principal respectivamente                                                 **
        *************************************************************************************'''
        super(MainWindow, self).__init__()
        self.setupUi(self)          # Prepara la ventana
        '''Declaración e inicialización de variables'''
        self.ref_wait = 0           # Variable para la pausa de tomado de referencia
        self.ref_control = 0        # Control referencia satisfactoria
        self.x_ref = 0              # Almacenamiento referencia wavelength
        self.y_ref = 0              # Almacenamiento referencia espectral power
        self.init = 1               # Indica que se ha iniciado la ventana
        self.name_index = 0         # Variable de identificador de archivo de medida
        self.status = 0             # Indicador de estado del programa
        self.param = []             # Almacen de parámetros de configuración
        self.measure_control = 0    # Control de bucle de medidas
        self.pausa = False          # Control estado de pausa entre medidas
        self.com, self.lista, self.osa = main.init_connection()# Clases necesarias de inicialización
        '''Conexionado entre señales y slots'''
        self.button_start.clicked.connect(self.button_start_clicked)    # Conecta botón de start
        self.button_config.clicked.connect(self.button_config_clicked)  # Conecta botón de
configuración
```

```
        self.button_readme.clicked.connect(self.button_readme_clicked)  # Conecta botón de
información
        self.button_connect.clicked.connect(self.button_connect_clicked)    # Conecta botón de
conectar a OSA
        self.button_pausa.clicked.connect(self.button_pausa_clicked)    # Conecta botón de pausa de
medida
        self.button_quit.clicked.connect(self.button_quit_clicked)      # Conecta botón de salir
        self.log_send = LogSignal()             # Genera objeto de la clase de conexión a la consola
        self.log_send.print_log.connect(set_log)# Conecta señal de actualización de consola de logs
        '''Preparación ventanas de medida'''
        self.fig1 = Figure()                                  # Genera contenedor medida de potencia
        self.fig2 = Figure()                                  # Genera contenedor mapa de colores
        self.spectral_power = self.fig1.add_subplot(111)      # Añade subplot medida de potencia
        self.birdview = self.fig2.add_subplot(111)            # Añade subplot mapa de colores
        self.canvas_measure = FigureCanvas(self.fig1)      # Establece canvas para el ploteo potencia
        self.measure_layout.addWidget(self.canvas_measure)  # Añade canvas al layout correspondiente
        self.canvas_measure.draw()                            # Muestra el contenedor
        self.canvas_birdview = FigureCanvas(self.fig2)
        self.birdview_layout.addWidget(self.canvas_birdview)
        self.canvas_birdview.draw()
        self.navi_toolbar = NavigationToolbar(self.canvas_measure, self)    # Genera la barra de
control de potencia
        self.measure_layout.addWidget(self.navi_toolbar)          # Lo añade al layout de potencia
        '''Preparación apariencia de la ventana principal de la interfaz'''
        self.setWindowTitle('PyOSAUI - jon.gmz.lbat@gmail.com') # Nombre de la ventana
        self.logoutput('text', 1)                          # Inicializa consola de logs
        self.text_status.setText('Status: Not connected')   # Estado inicial del software
        self.button_pausa.setVisible(False)                # Desactiva el boton de pausa
        icon_pix = QtGui.QPixmap('icon.png')               # Genera mapa de pixeles para el icono del
botón de start
        icon = QtGui.QIcon(icon_pix)                       # Genera el icono del botón de start
        self.button_start.setIcon(icon)                    # Establece el icono generado
        self.button_start.setStyleSheet('color : green')    # Establece el color del botón de start
        self.bitmap = QtGui.QBitmap('back.png')            # Genera el mapa de bits para la forma de la
ventana
        self.region = QtGui.QRegion(self.bitmap)           # Genera la región a partir del mapa de bits
        self.setMask(self.region)                          # Establece la región generada como máscara
para la ventana
        self.palette = QtGui.QPalette()                    # Inicia un objeto de tipo QPallete para el
fondo de la interfaz
        self.palette.setBrush(QtGui.QPalette.Background,
                              QtGui.QBrush(QtGui.QPixmap("wall.jpg")))  # Prepara el objeto para el
fondo
        self.setPalette(self.palette)                      # Establece el fondo de la ventana
        '''Preparar directorio de guardado de medidas y redirigir al programa'''
        self.dir = './Medidas/' + strftime('%d_%m_%y')  # Genera el directorio de guardado
        if not os.path.exists(self.dir):                   # Crea el directorio si no existe
            os.makedirs(self.dir)
        os.chdir(self.dir)                                 # Redirige al programa a dicho directorio
        self.show()  # Muestra la ventana
```

```
    def get_index(self, next):
        '''Obtiene el identificador del archivo de guardado'''
        s = []                                          # Inicializa variable s
        add = 0
        for f in glob.glob("*.mat"):                    # Detecta el último identificador
            for a in f.split():
                if a.isdigit(): s.append(a)
        for item in s:
            if int(item) > self.name_index: self.name_index = int(item)
            add = 1
        if next == 1 and add == 1: self.name_index += 1     # Aumenta en 1 el identificador si se
solicita

    def mouseMoveEvent(self, event):
        '''La definición de este evento junto con el "mousePressEvent" permite mover la interfaz
        arrastrando desde cualquier punto'''
        if event.buttons() == QtCore.Qt.LeftButton:
            self.move(event.globalPos().x() - self.drag_position.x(),
                      event.globalPos().y() - self.drag_position.y())
        event.accept()

    def mousePressEvent(self, event):
        if event.button() == QtCore.Qt.LeftButton:
            self.drag_position = event.globalPos() - self.pos()
        event.accept()

    def logoutput(self, text, define):
        '''Define y actualiza la consola de logs'''
        if define == 1:
            self.front_color = QtGui.QColor(255, 255, 255)      # Genera color para la letra
            self.log_console.setReadOnly(True)                  # Establece propiedad solo lectura
            self.log_console.setLineWrapMode(QtGui.QTextEdit.NoWrap)
            self.log_console.setStyleSheet("background-color: black")    # Color del fondo
        else:
            self.log_console.setTextColor(self.front_color)     # Establece el color a fuente
            self.font = self.log_console.font()
            self.font.setFamily('Courier')                      # Establece el tipo de fuente
            self.font.setPointSize(8)
            self.log_console.setCurrentFont(self.font)          # Actualiza la fuente en el objeto
            self.log_console.insertPlainText(text)              # Inserta el texto enviado
        sb = self.log_console.verticalScrollBar()        # Siempre muestra la ultima línea de texto
        sb.setValue(sb.maximum())
        self.log_console.moveCursor(QtGui.QTextCursor.End)

    def return_measure(self):
        '''Comprueba el modelo y llama a la función de medida correspondiente. Devuelve los vectores
de medida'''
        if self.connect_dialog.osa_selected == 0:
            '''EXFO'''
            # Llama a la función de medida EXFO
            x, y = main.measure_loop_exfo(self.connect_dialog.instr,
```

```
                                        self.config_dialog.osa_measure,
                                        self.config_dialog.param)
                self.x_lim_inf = self.param[1]
                self.x_lim_sup = self.param[2]
            else:
                if self.connect_dialog.osa_selected == 1:
                    '''AGILENT'''
                    # Llama a la función de medida AGILENT
                    x, y = main.measure_loop_agilent(self.connect_dialog.instr,
                                                     self.config_dialog.osa_measure,
                                                     self.config_dialog.param)
                    self.x_lim_inf = self.param[4] - self.param[5]/2
                    self.x_lim_sup = self.param[4] + self.param[5]/2
                else:
                    x = 0
                    y = 0
            return x, y

        def guarda_ref(self):
            '''Almacena la referencia'''
            nombre_fichero = QtGui.QFileDialog.getSaveFileName(self, "Guardar referencia")  # Dialogo de
guardado
            if nombre_fichero:
                ref_path = QtCore.QFileInfo(nombre_fichero).path()                # Captura ruta
                name_xref = ref_path + '/' + nombre_fichero + 'x_ref.txt'         # Nombre de guardado x
                name_yref = ref_path + '/' + nombre_fichero + 'y_ref.txt'         # Nombre de guardado y
                main.np.savetxt(name_xref, self.x_ref, delimiter=' ')           # Guarda el archivo x_ref
                main.np.savetxt(name_yref, self.y_ref, delimiter=' ')           # Guarda el archivo y_ref

        def carga_ref(self):
            '''Carga una referencia'''
            nombre_xref = QtGui.QFileDialog.getOpenFileName(self, 'Cargar referencia longitudes de
onda')
            if nombre_xref:
                ref_name_xref = QtCore.QFileInfo(nombre_xref).fileName()            # Nombre fichero
                ref_path_xref = QtCore.QFileInfo(nombre_xref).path()                # Ruta fichero
                if ref_name_xref.find('.txt') is not -1:
                    nombre_yref = QtGui.QFileDialog.getOpenFileName(self, 'Cargar referencia potencia
espectral')
                    if nombre_yref:
                        ref_name_yref = QtCore.QFileInfo(nombre_yref).fileName()
                        ref_path_yref = QtCore.QFileInfo(nombre_yref).path()
                        if ref_name_yref.find('.txt') is not -1:     # Si es correcto se cargan las ref.
                            name_xref = ref_path_xref + '/' + ref_name_xref
                            name_yref = ref_path_yref + '/' + ref_name_yref
                            try:
                                self.x_ref = main.np.loadtxt(name_xref, delimiter=' ')
                                self.y_ref = main.np.loadtxt(name_yref, delimiter=' ')
                                self.ref_control = 1
                            except:
                                text = 'Error: valores no encontrados'
```

```
                        window.log_send.emit_log(text)
                else:
                    text = 'Error: archivo yref incorrecto'
                    window.log_send.emit_log(text)

        else:
            text = 'Error: tipo archivo incorrecto'
            window.log_send.emit_log(text)

def button_quit_clicked(self):
    '''Cierra la interfaz ante una petición de cierre desde el botton "button_quit"'''
    self.close()

def button_connect_clicked(self):
    '''Slot para el botón de conexión, crea un diálogo para la conexión'''
    if self.status == 3:                    # Comprueba el estado del programa
        text = 'Error: Medida en curso\n'
        self.logoutput(text, 0)
        msg_box = QtGui.QMessageBox()
        msg_box.setWindowTitle('Error')
        msg_box.setText('Pare la medida primero')
        msg_box.exec_()
    else:
        self.connect_dialog = ConnectDialog(self.com, self.lista, self.osa) # Crea el objeto
        for item in self.lista:             # Configura la ventana
            self.connect_dialog.lista_principal.addItem(item)
        self.connect_dialog.text_cabecera.setText('Seleccione conexión: ')
        self.connect_dialog.show()          # Muestra la ventana

def button_config_clicked(self):
    '''Slot para el botón de configuración, crea un dialogo de configuración'''
    if self.status == 0:                    # Comprueba el estado del programa
        text = 'Error: Conexión no encontrada\n'
        self.logoutput(text, 0)
        msg_box = QtGui.QMessageBox()
        msg_box.setWindowTitle('Error')
        msg_box.setText('Establezca primero una conexión.')
        msg_box.exec_()
    else:
        if self.status == 3:
            text = 'Error: Medida en curso\n'
            self.logoutput(text, 0)
            msg_box = QtGui.QMessageBox()
            msg_box.setWindowTitle('Error')
            msg_box.setText('Pare la medida primero')
            msg_box.exec_()
        else:
            self.config_dialog = ConfigDialog() # Crea el objeto
            self.config_dialog.show()           # Muestra la ventana
```

```
    def button_pausa_clicked(self):
        '''Modifica el valor de la variable pausa, que se evalúa en el bucle de medidas en el thread
de medida'''
        self.pausa = not self.pausa
        if self.pausa:
            text = 'Medida pausada\n'
            self.logoutput(text, 0)
            self.button_pausa.setText('Continuar')
        else:
            text = 'Medida en curso\n'
            self.logoutput(text, 0)
            self.button_pausa.setText('Pausa')

    def button_readme_clicked(self):
        '''Muestra información al usuario'''
        splash.show()
        pass

    def button_start_clicked(self):
        '''*****************************************************************************************
        **Método para la respuesta a una petición de comenzar la medida (o pararla)             **
        *****************************************************************************************'''
        if self.status == 2:
            msg_box = QtGui.QMessageBox()                   # Genera un objeto de tipo QMessageBox
            msg_box.setWindowTitle('Gestión de referencias')# Configura la ventana de mensaje
            msg_box.setText('¿Desea tomar referencias?')
            btnQS = QtGui.QPushButton('Realizar referenciado')
            msg_box.addButton(btnQS, QtGui.QMessageBox.YesRole)
            btnNo = QtGui.QPushButton('Cargar una referencia')
            msg_box.addButton(btnNo, QtGui.QMessageBox.NoRole)
            ret = msg_box.exec_()
            if ret == 0:                        # Se ha elegido realizar un referenciado
                self.ref_wait = 1               # Indica al programa que debe esperar al referenciado
                self.ref_thread = ReferenceThread()  # Genera el hilo de ejecución de referencia
                self.ref_thread.start()             # Inicia el thread
                while self.ref_wait:                # Espera a realizar el referenciado
                    main.sleep(0.01)
                text = 'Referencia tomada\n'        # Actualiza consola de logs
                self.logoutput(text, 0)
                self.ref_control = 1
                msg_box2 = QtGui.QMessageBox()      # Genera diálogo para guardar la referencia
                msg_box2.setWindowTitle('Guardar referencia')
                msg_box2.setText('Se ha tomado la referencia, desea guardarla?')
                btnQS = QtGui.QPushButton('Si')                                     # Botón si
                msg_box2.addButton(btnQS, QtGui.QMessageBox.YesRole)
                btnNo = QtGui.QPushButton('No')                                     # Botón no
                msg_box2.addButton(btnNo, QtGui.QMessageBox.NoRole)
                ret = msg_box2.exec_()
                if ret == 0:                        # Se ha elegido guardar la referencia
                    '''GUARDA REF'''
                    self.guarda_ref()               # Llama a la función de guardado
```

```
                text = 'Referencia guardada\n'          # Actualiza consola de logs
                self.logoutput(text, 0)
            else:
                '''CARGA REFERENCIA'''                  # Se ha elegido cargar una referencia
                self.carga_ref()                        # Función de carga
                text = 'Referencia cargada\n'           # Actualiza consola de logs
                self.logoutput(text, 0)
            '''BUCLE MEDIDAS'''
            if self.ref_control:                        # Comprueba que haya una referencia disponible
                self.measure_control = 1
                self.status = 3
                self.text_status.setText('Midiendo') # Actualiza estado del software
                self.button_start.setText('Parar')# Cambia cadena de caracteres del botón de medida
                self.button_start.setStyleSheet('color : red')
                self.button_pausa.setVisible(True)
                self.measure_loop = MeasureThread()                 # Genera un hilo de ejecución de
medida
                self.measure_loop.start()                           # Inicia el thread
                text = 'Proceso de medición iniciado\n'             # Actualiza consola de logs
                self.logoutput(text, 0)
            else:
                text = 'Error en la referencia'             # No hay una referencia disponible
                self.logoutput(text, 0)


        else:
            if self.status == 3:                            # Se ha pulsado el botón de parar medida
                self.get_index(1)
                self.button_pausa.setVisible(False)         # Se desactiva el botón de pausa
                self.measure_control = 0                    # Variable control de medida
                self.pausa = False                          # No puede haber pausa si no hay medida
                self.status = 2                             # Actualiza estado del software
                self.button_start.setText('Medir')          # Cambia botón de medida
                self.button_start.setStyleSheet('color : green')
                self.spectral_power.clear()                 # Limpia las ventanas de visualización
                self.canvas_measure.draw()
                self.birdview.clear()
                self.canvas_birdview.draw()
                self.text_status.setText('Status: Connected to ' +

self.connect_dialog.lista[self.connect_dialog.conn_selected] + '; Using ' +

self.connect_dialog.osa.lista[self.connect_dialog.osa_selected])
            else:
                text = 'Error: Medida no configurada\n'     # Avisa al usuario si se solicita una
medida sin configurar primero el analizador
                self.logoutput(text, 0)
                msg_box = QtGui.QMessageBox()
                msg_box.setWindowTitle('Error')
                msg_box.setText('Configure primero una medida.')
                msg_box.exec_()
```

```
if __name__ == '__main__':
    '''Ejecución del software'''
    app = QtGui.QApplication(sys.argv) # Genera la aplicación sobre la que se construirá la interfaz
    os.chdir('Resources')              # Redirige el programa al directorio de recursos
    splash_pix = QtGui.QPixmap('splash_screen.png') # Prepara el mapa de pixeles para generar el
splash
    splash = QtGui.QSplashScreen(splash_pix, QtCore.Qt.WindowStaysOnTopHint) # Genera el splash
    splash.setMask(splash_pix.mask())               # Establece la forma del splash como máscara
    splash.show()                                   # Muestra el splash
    app.processEvents()                             # Comienza a procesar eventos de ejecución
    window = MainWindow()                           # Genera el objeto de la ventana principal
    text = 'Modulos cargados correctamente\nListo para comenzar\n' # Actualiza consola de logs
    window.log_send.emit_log(text)
    splash.finish(window)           # Cierra automaticamente el splash cuando termina de cargar todo
    sys.exit(app.exec_())           # ejecuta la aplicación
```